STalker™ 3

GEM Based Telecommunications Software
for the Atari ST/TT and Compatibles

GRIBNIF SOFTWARE

STalker 3

Atari GEM Based Telecommunication

Gribnif Software

SER 0007666
VER 3.04

STalker 3

Programs Copyright © 1988-1993 by Strata Software
Documentation & Packaging Copyright © 1991-1993 by Gribnif Software
All Rights Reserved

Enclusive World Wide Marketing and Distribution by Gribnif Software

STeno, STalker, and BackTALK are trademarks of Strata Software. NeoDesk is a registered trademark of Gribnif Software. All other trademarks belong to their respective holders.

No part of this book, disk, and/or software may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage or retrieval system, without the written permission of the publisher.

Program written and designed by Eric Rosenquist
Manual written by Bob Goff & Mike Cohan
Manual typeset by Rick Flashman
Manual edited by Dan Wilga & Rick Flashman

Gribnif Software
P.O. Box 779,
Northampton, MA 01061
U.S.A.

Tel: (413) 247-5620
Fax:(413) 247-5622

Printed in the United States of America
Second Edition

1. Table of Contents

1.. Preface 1
1.1.. Thank You 1
1.2.. Software Piracy 1
1.3.. Copyright Notice 1
1.4.. Licensing Agreement 1
1.4.1.. If you DO NOT AGREE 2
1.4.2.. If you DO AGREE 2
1.5.. Disclaimer Notice 3
1.6.. Defective Disks 3
1.7.. Upgrading STalker 3

2.. Introduction 4
2.1.. Program Overview 4
2.2.. About This Manual 5
2.3.. Problems and Support 5

3.. Getting Started 7
3.1.. Registering and Backing Up the Disk 7
3.2.. Installing STalker 3 8
3.2.1.. To install as a desk accessory 8
3.2.2.. To install as a GEM program 9

4.. Running STalker 3 11
4.1.. Running as a GEM Program 11
4.2.. Running as a Desk Accessory 11
4.3.. Running as a NeoDesk Accessory 11

5.. Operating STalker 3 12
5.1.. Introduction 12
5.2.. Dialog Boxes 12
5.3.. Menu Bar 14
5.3.1.. Running as a program 14
5.3.2.. Running as a desk accessory 15
5.3.3.. Keyboard shortcuts 15
5.4.. Window Controls 15
5.4.1.. Title bar (shows Status Line or Menu Bar) 16
5.4.2.. Moving the window 16
5.4.3.. Close button 17
5.4.4.. Full Size button 17

5.4.5.. Resize gadget 18
5.4.6.. Scroll Bars 19
5.4.7.. Data region and virtual screen 20
5.4.8.. Cut and paste/block functions 20
5.4.9.. Selecting a block 21
5.4.10.. Block Action Window options 21

6.. Introduction to Telecommunications 23
6.1.. Introduction 23
6.2.. Connecting Computers Together 23
6.2.1.. Modems and null modem cables 23
6.2.2.. Hayes and Hayes compatible modems 24
6.2.3.. High speed modems 24
6.3.. Configuration Definitions 24
6.3.1.. Baud rate 24
6.3.2.. Full/Half Duplex 25
6.3.3.. Word length 25
6.3.4.. Stop Bits 26
6.3.5.. Parity 26
6.3.6.. Flow Control 26
6.3.7.. Terminal Emulation 26
6.4.. How to Connect to a Host System 27
6.4.1.. Set configuration, terminal settings, etc. 27
6.4.2.. Dialing 27
6.4.3.. Hanging up 28
6.5.. File Transfer 28
6.6.. Basics of Uploading/Downloading 28
6.6.1.. Text (ASCII) Files 28
6.6.2.. Binary Files 29
6.6.3.. Binary file transfer protocols 31

7.. Drop-Down Menus and Dialog Boxes 33
7.1.. Desk (or Atari Fuji) Drop-Down Menu 33
7.1.1.. STalker info...[Alt][I] 33
7.1.2.. Desk accessories 33
7.2.. File Drop-Down Menu 34
7.2.1.. Open configuration file... 34
7.2.2.. Save configuration [Alt][G] 34
7.2.3.. Save configuration as... [Shift][Alt][G] 34
7.2.4.. Send text file... [Alt][S] 35
7.2.5.. Receive text file... [Alt][R] 35
7.2.6.. Upload binary file... [Alt][U] 36
7.2.7.. Download binary file... [Alt][D] 37
7.2.8.. Invoke STeno [Alt][Tab] 37

7.2.9.. Remote (mini-BBS) mode... [Alt][B] 38
7.2.10.. Shadow access... [Alt][=] 40
7.2.11.. Quit/Close window [Alt][Q] 40
7.3.. Edit Drop-Down Menu 41
7.3.1.. Abort current operation [Undo] 41
7.3.2.. Copy window to clipboard [Alt][C] 41
7.3.3.. Paste clipboard (text) [Alt][V] 41
7.3.4.. Paste clipboard (binary) [Alt][Shift][V] 41
7.3.5.. Clear window contents [Control][Clr/Home] 42
7.4.. Settings Drop-Down Menu 42
7.4.1.. Communications settings... [Alt][P] 42
7.4.2.. File transfer settings... [Alt][X] 43
7.4.2.1.. Text settings 43
7.4.2.2.. Binary File protocol settings 44
7.4.2.3.. Setting X/Y/ZModem Upload/Download Parameters 45
7.4.3.. Terminal settings... [Alt][T] 47
7.4.4.. Communication port... [Shift][Alt][P] 48
7.4.5.. Function key settings... [Alt][F] 49
7.4.6.. Modem Settings... [Alt][M] 50
7.4.7.. Preference Settings... [Alt][Y] 52
7.5.. Options Drop-Down Menu 54
7.5.1.. Capture Mode [Alt][O] 54
7.5.2.. Echo Typed Characters [Alt][E] 55
7.5.3.. Disable STalker [Alt][K] 55
7.5.4.. Reset Communication Port [Alt][Esc] 55
7.5.5.. Send a BREAK Signal [Alt][Backspace] 56
7.5.6.. Send Text to Printer [Alt][L] 56
7.5.7.. Show Control Characters [Alt][Return] 56
7.5.8.. Window style 57
7.6.. Dial Drop-Down Menu 57
7.6.1.. Autodialer Directory [Alt][A] 57
7.6.2.. Dial Selected Entries... [Shift][Alt][A] 61
7.6.3.. Hang Up Phone Line... [Alt][H] 61
7.6.4.. Hang-up then Dial Next [Shift][Alt][H] 61
7.6.5.. Reset $/Time Clock [Control][Shift][Alt][$] 62
7.6.6.. Dialing Groups [Alt][0] through [Alt][9] 62
7.7.. BackTALK Drop-Down Menu 62
7.7.1.. Run BackTALK Script [Alt][Enter] 62
7.7.2.. Change Menu Entries... [Alt][.] 63
7.7.3.. Scripts[Alt][0] through [Alt][9] 63
7.8.. Help Menu 64

8.. Using STeno with STalker 65
8.1.. Keeping Multiple Copies of STalker and STeno in Memory . 65
8.1.1.. Example 66
8.2.. Using STeno as a Type-ahead Buffer for STalker 67

9.. Using STalker 3 with NeoDesk 69

10.. Technical Information 70
10.1.. Shutdown 70
10.2.. The STalker Clipboard 70
10.3.. Y/Z Modem Information 70
10.4.. Timestamps in Y/ZModem 71
10.5.. File Transfer Efficiency 71
10.6.. Capture Mode 71
10.7.. Reset Communication Port 72
10.8.. Autodialing Routine 72
10.9.. ASCII Null Character 74
10.10.. Using MultiDesk 74
10.11.. Using the FastLoad Option 74
10.12.. Using PinHead 74
10.13.. Using GDOS 74
10.13.1.. Sample ASSIGN.SYS File 75

11.. Index 76

1. Preface

1.1. Thank You

All of us at Gribnif Software would like to thank you for licensing this copy of *STalker™ 3*. We have worked very hard to make it what it is today, and we hope that you appreciate the results. We want you to know that we appreciate your support.

1.2. Software Piracy

A large amount of our time and effort has gone into making this *STalker 3* package possible. Therefore, we ask that you not give copies of this program to your friends. Not only is this illegal, but in the end it hurts not just us, but you.

The loss of sales due to piracy will only make it harder for us to develop new and exciting Atari software like STalker. We have many ideas for new Atari software development, but whether they come to pass depends on you and other Atari owners. *Please, do not give out copies of our software!*

1.3. Copyright Notice

The *STalker 3* program is copyrighted by Strata Software, which is represented by Gribnif Software. Gribnif Software (hereby referred to as "we" or "us") also holds all other copyrights over the program documentation and packaging. We reserve all rights to and ownership of this package, its disk, software, and documentation. Use of this software, disk, and manual are subject to the terms of the following licensing agreement.

1.4. Licensing Agreement

In agreeing to the terms of the licensing agreement for this *STalker 3* package (which consists of the disk, the software on the disk, the documentation, and any other material contained in the package), you are licensing this package for your own personal use for as long as you do not violate this licensing agreement. You may make backup copies of the software for your own personal use and/or for archival storage only. The *STalker 3* software may only be used by a single person on one or more computers, or on a single machine by one or more persons. This package cannot be licensed to more than one user at a time. Please contact us for special multiple user and multiple machine licensing agreements.

Renting or lending this package or any portion of it, placing it on a BBS (Bulletin Board System), reproducing the package or any portion of it in any way which is not permitted under the first paragraph of this licensing agreement, and/or sharing the package or any portion of it with other people is not allowed and will result in the automatic termination of this licensing agreement. In other words, *you are not allowed to distribute copies of this program or its documentation in any way to anyone.*

You should remember that distribution of copies of this program and/or its documentation without proper permission violates federal and international copyright laws, and we will prosecute illegal users to the full extent of the law.

To conform to the terms of this licensing agreement, you agree to correctly fill out and mail the enclosed *STalker 3 Warranty/Upgrade Card.* We must have this card on file before we can supply you with any technical support for STalker 3.

You also agree to correctly register the *STalker 3* program as instructed in this manual, and to correctly enter your name and address. Once this copy of *STalker 3* is registered, you agree to destroy all non-registered copies of *STalker 3* you may have. Keeping a non-registered copy of *STalker 3* after registering a copy is a violation of this licensing agreement.

If this licensing agreement is violated by you, you agree to mail to Gribnif Software your copy of the *STalker 3* package (including all of its contents), and destroy any copies you may have, paying all expenses. Notification of the violation is not required. Remember, *STalker 3* is our property, you are simply licensing its use.

If you violate this licensing agreement, you also agree to pay Gribnif Software for any damages that occur. It is your responsibility to uphold this licensing agreement.

You also agree that you have read, understood, and agree to the **Copyright Notice** and **Disclaimer Notice** as they appear in this manual.

1.4.1. If you DO NOT AGREE

If you do not agree with the terms and conditions of this licensing agreement, do not unseal the disk envelope holding the program disk. Please return this *STalker 3* package to your place of purchase, or contact us directly. When you return the *STalker 3* package, you must include all of the original documentation and packaging, and the disk envelope must still be sealed and unopened, and the program must not be registered.

1.4.2. If you DO AGREE

If you agree with the terms and conditions of this licensing agreement and you wish to use *STalker 3*, open the disk envelope. Opening the disk envelope is considered an acceptance by you of all the stipulations presented in this licensing agreement.

You also agree to correctly fill out and mail the enclosed *STalker 3 Warranty/ Upgrade Card*. We must have this card on file before we can supply you with any technical support for *STalker 3*.

You also agree to register *STalker 3* using the procedure documented in this manual, and to correctly enter your name and address during this process.

1.5. Disclaimer Notice

Gribnif Software makes no warranties, either expressed or implied, with respect to this manual or the software described in this manual and supplied in the package, its quality, performance, merchantability, or fitness for any particular purpose. *STalker 3* is licensed 'as is'.

1.6. Defective Disks

If the *STalker 3* program disk is damaged or proves defective, you should contact us directly for a replacement. Do not return the damaged disk to the place of purchase, since they are not authorized to repair, replace, or swap it.

1.7. Upgrading STalker

From time to time we may come out with new versions of *STalker*. You should contact us directly for upgrade information. Dealers are not authorized to upgrade the program.

2. Introduction

2.1. Program Overview

STalker 3 is a full featured terminal emulator, complete with file transfer capabilities. It fully supports the features that make your Atari so easy to use, including windows, drop-down menus, and dialog boxes. And best of all, it operates as a desk accessory, so all of STalker 3's features are at your fingertips even when you are running other GEM programs! Of course, you can also run STalker 3 as a stand-alone program. Some important features of STalker 3 are:

- Runs inside a GEM window with full GEM support
- Operates as a desk accessory or a stand-alone program
- Full background operation including file transfer - send or receive data while you use another GEM based program!
- Cut and Paste, via Atari's text clipboard
- VT-52, VT-100, and PC-ANSI emulation
- ASCII, XModem, XModem 1K, YModem, YModem-G, ZModem, and CompuServe B+ file transfers
- Programmable Autodialer with up to 30 numbers
- 20 user-definable function keys
- Virtual screen size of 24-9999 lines by 40-200 columns
- GDOS support for different terminal fonts and sizes
- Support for NeoDesk® 3 and NeoDesk® 4 - Gribnif Software's desktop replacements
- Fully integrated with STeno™ - Gribnif Software's GEM based text editor
- Includes BackTALK™ - STalker's powerful scripting language

2.2. About This Manual

This manual tells you everything you need to know in order to operate STalker on any Atari compatible computer. Although you will find STalker to be easy and intuitive to use, if you do not read the entire manual, you may miss out on a number of powerful features.

If any changes or corrections have been made to the program or manual since the printing of this manual, you will find a file on the disk named READ_ME.NOW. Open this file (double-click on it from the desktop, then follow the prompts) to display its contents. Press the space bar to read the document screen by screen. You may also wish to print this file for reference, since it may contain important changes or additions to this manual and/or program.

2.3. Problems and Support

Gribnif Software has tried very hard to make this software package as free from defects as possible. But, like all computer software, it is never quite finished, so be sure to mail your warranty card to us so we can notify you of new versions of STalker as they become available. We must have your warranty card on file before we can make available to you any technical support or upgrades.

If you have any problems with STalker please double check:

- The READ_ME.NOW file on the STalker 3 program disk.
- This manual.

About 90% of the questions we receive are easily answered by looking in the manual or READ_ME.NOW file. If, after a careful reading, you are still having problems, or have questions about STalker 3 not covered by the documentation, you can reach our technical support line at (413) 247-5620 during normal business hours (Eastern Standard Time). When you call, be sure to have your serial number handy.

It is easier for us to answer your questions over the phone if you are in front of the computer and running STalker during the call. It is also helpful if you can duplicate the problem.

You can also write to us at Attn: STalker Tech Support, Gribnif Software, P.O. Box 779, Northampton, MA 01061, U.S.A., or you can send us a fax at (413) 247-5622. We will try to answer as quickly as possible.

If you write or fax, be sure to include a note with the following information:

1. Your name.
2. Your address.
3. A daytime telephone number, if available.
4. Your fax number, if applicable.
5. Your STalker 3 serial number.
6. Your STalker 3 program version (check under the *STalker Info...* menu entry in the Atari "Fuji" Logo drop-down menu).

We are also available on the General Electric Network of Information Exchange (GEnie). You can contact us there under the Atari roundtable on page 475, category 17. Electronic mail can be sent to our GRIBNIF address on GEnie.

On CompuServe you can find us in the Atari Vendor area (GO ATARIVEN) under Gribnif Software, category 8. Electronic mail can be sent to our 75300,1131 address there. If you have access to the Internet you can send electronic mail to our CompuServe account at "75300.1131@compuserve.com"

As before, please include your STalker 3 serial number with any inquiries.

3. Getting Started

3.1. Registering and Backing Up the Disk

Before you do anything else, you must register your original STalker 3 Master Disk so that it becomes operational. STalker 3 is not copy protected. Instead, we ask that you follow this simple registration procedure which will ask you for your legal name and address (needed in order for you to receive help or updated versions of STalker 3, and to conform with the terms and requirements of our licensing agreement) so that your copy of STalker 3 becomes fully operational. Remember, registering STalker 3 is part of the licensing agreement as stated in the previous pages. As with any contract, make sure you have read the licensing agreement, fully understand it, and agree with it.

Follow these easy steps to register your copy of STalker 3. Do NOT write-protect the master disk or remove the disk during the procedure, because the registration program needs to modify the contents of the disk. You must follow these instructions or STalker 3 will not work.

1. Insert your original STalker 3 master disk into disk drive A and turn on your computer.
2. You should see the REGISTER.PRG icon in an open window. If no window is visible, open a window to drive A and look for the REGISTER.PRG icon.
3. Run the REGISTER.PRG program by opening it or double-clicking on it.
4. Follow the on-screen instructions. At the prompt, enter your legal name and address.
5. Wait a couple of seconds. The REGISTER.PRG will run and register your personal licensed copy of STalker 3.

STalker 3 will NOT work if it has not been registered. The name and address you enter must match the information you filled out on the STalker 3 warranty card. If you move, please mail us a U.S. Post Office Change of Address Card or equivalent with your new address. Be sure to include your STalker 3 serial number.

If your original STalker 3 master disk does not have a REGISTER.PRG program on it, you may have been sold a previously registered copy of STalker 3. Please contact us for a replacement, as nobody likes buying used software.

After registering your copy of STalker, you should make a backup copy of the STalker program disk. Floppy disks are magnetic media and you never know what can happen to them. For this reason, we do not recommend using the original STalker program disk for daily use. Instead, we STRONGLY recommend that you make a backup of the program disk and install STalker on your hard drive or regular boot disk.

The desktop disk copy procedure, as documented in your desktop manual, will allow you to make a backup of the STalker program disk.

3.2. Installing STalker 3

As STalker is not copy protected, it can be copied to your regular boot disk or hard drive. When run as a stand-alone program, STalker can be stored in any folder or directory. When run as a desk accessory, it must be stored in the same directory where your other desk accessories are located. Be certain to copy the STALKER.RSC file into this directory, along with the STALKER.PRG or STALKER.RSC file, or STalker will be unable to run.

Note: The STALKER.PRG and STALKER.ACC files are actually the same file. Just rename the file ending to change the way it works.

If you find the previous explanation slightly confusing, the following installation procedure will simplify the process.

3.2.1. To install as a desk accessory

- Hard Drive users

1. Place the STalker program disk in disk drive A. Then open a desktop window to drive A.
2. Open a second window to the boot partition of your hard drive (usually drive C). A *boot partition* is the partition from which all of your accessories normally load.
3. Select and drag the STALKER.ACC and STALKER.RSC files from the STalker program disk window (the drive A window) to the hard drive window (the drive C window). This will copy the STalker program to your hard drive.

Note: The end goal is to place STALKER.ACC and STALKER.RSC in the same directory as the rest of your desk accessories.

- Floppy Drive users

1. If you presently do not have a boot disk, then format a new disk. This will become your boot disk. A *boot disk* is a disk that the computer uses when you turn it on to initialize itself and to load desk accessories. From now on, this disk should go in drive A whenever you turn on or reboot (reset) your computer.

2. If you have a single drive system, place your boot disk in drive A. If you have a double drive system, place the boot disk in drive B.

3. Open a window to drive B by double clicking on the drive B icon. If you have a single drive system, press the [Return] key at the prompt asking you to insert disk B into drive A. From this point on, your boot disk is considered "Disk B".

4. Place the STalker program disk in drive A. If you have a single drive system, remove your boot disk before inserting the STalker program disk.

5. Open a window to drive A by double clicking on the drive A icon. If you have a single drive system, press the [Return] key at the prompt asking you to insert disk A into drive A. From this point on, your STalker program disk is "Disk A".

6. You will now have two windows open, one labeled "A" and the other labeled "B". Drag the STALKER.ACC file and the STALKER.RSC file from the "A" window (the STalker program disk window) over to the "B" window (the boot disk window). This will copy the STalker program to your boot disk.

Note: If you have a single drive system, the operating system will ask you for a series of disk swaps. These messages will prompt you at different times to insert disk A (the STalker program disk) or disk B (your boot disk) into drive A. Carefully follow these prompts and be certain to insert the correct disk so that all the STalker files are copied correctly.

3.2.2. To install as a GEM program

- Hard Drive users

1. Place the STalker program disk in disk drive A. Then open a desktop window to drive A by double-clicking on the drive A icon.

2. Open a second window to your hard drive. It does not matter which hard drive partition (letter) you choose, since you can place the STalker program anywhere.

3. Create a new folder with the name STALKER on the hard drive. This is done from the desktop by selecting *Create Folder...* or *New Folder...* from the **File** drop-down menu.

4. Open the hard drive window to the newly created STALKER folder by clicking on the STALKER folder icon within the hard drive window.

5. Select and drag the STALKER.PRG and STALKER.RSC files from the STalker program disk window (the drive A window) to the hard drive window. This will copy the STalker program to your hard drive.

Note: The end goal is to place STALKER.PRG and STALKER.RSC in the STALKER folder you created in your hard drive.

- Floppy Drive users

1. Select the disk you wish to place the STalker program on. If you do not have enough space on any of your disks, then format a new disk. This will be known as the "new STalker disk".

2. If you have a single drive system, place the new STalker disk in drive A. If you have a double drive system, place the new STalker disk in drive B.

3. Open a window to drive B by double clicking on the drive B icon. If you have a single drive system, press the [Return] key at the prompt asking you to insert disk B into drive A. From this point on, the new STalker disk is considered "Disk B".

4. Place the original STalker program disk in drive A. If you have a single drive system, remove the new STalker disk before inserting the original STalker program disk.

5. Open a window to drive A by double clicking on the drive A icon. If you have a single drive system, press the [Return] key at the prompt asking you to insert disk A into drive A. From this point on, your original STalker program disk is considered "Disk A".

6. You will now have two windows open, one labeled "A" and the other labeled "B". Drag the STALKER.PRG file and the STALKER.RSC file from the "A" window (the original STalker program disk window) over to the "B" window (the new STalker disk window). This will copy the STalker program to the new STalker disk.

Note: If you have a single drive system, the operating system will ask you for a series of disk swaps. These messages will prompt you at different times to insert disk A (the original STalker program disk) or disk B (the new STalker disk) into drive A. Carefully follow these prompts and be certain to insert the correct disk so that all the STalker files are copied correctly.

4. Running STalker 3

4.1. Running as a GEM Program

You can run STalker as a stand alone GEM program by double-clicking on the STALKER.PRG icon. The STalker program will load into memory and begin to run.

4.2. Running as a Desk Accessory

If you have installed STalker as a desk accessory, pressing the reset button on the computer (with your boot disk in drive A for floppy based systems) will cause STalker to load and install itself in memory.

Once STalker is loaded, you can access it by moving the mouse to the Desk or Atari Logo drop-down menu (this is the leftmost menu) and selecting the STalker : Off-line entry. (The "Off-line" part of the entry will change as you use STalker.)

When STalker is running as a desk accessory, it is available inside any regular GEM program that lets you access the desk accessory menu. Some programs (usually those ending in .TOS or .TTP) do not allow access to desk accessories until you have exited them.

The most convenient feature of running STalker as a desk accessory is that it never stops running as long as you are using GEM based programs. For instance, you can begin downloading a file from a BBS, then close the STalker window and use another GEM based program, and the downloading will continue in the background.

Note: A common mistake people make when first using STalker is to have the desk accessory version of STalker in memory (you will see "STalker: Off-line" under the Desk menu) at the same time that STalker is running as a program. Doing this will cause both copies of STalker to read data from the communications port at the same time, and can lead to unpredictable results when trying to use STalker's features. The solution is to make sure that the desk accessory version of STalker is not present when you want ot run STalker as a program.

4.3. Running as a NeoDesk Accessory

You can install STalker as a *NeoDesk Accessory*. These are special desk accessories which can communicate with Gribnif Software's NeoDesk 3 and NeoDesk 4 desktop replacements. This gives the STalker accessory some extra capabilities. For more information, check the *Using STalker with NeoDesk* section of this manual.

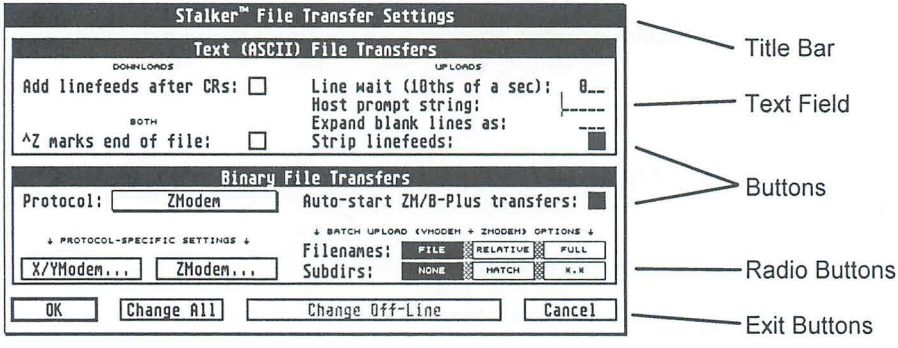
5. Operating STalker 3

5.1. Introduction

For the most part, STalker works just like any other GEM program or desk accessory, and is completely consistent with the GEM program design guidelines. This makes it easier for you to learn how to use STalker because you can apply the knowledge you have acquired from using other GEM programs.

5.2. Dialog Boxes

When you issue a command to STalker, it will sometimes need to ask you for information. When this happens, STalker will display a Dialog Box on the screen. A dialog box has a title bar at the top, followed by a series of "video buttons" and/or text fields.



Video buttons display various options available, while text fields are spaces into which you can enter information.

When video buttons are used in a dialog box, you will notice that some of them may be displayed in reverse video (white letters on a black background). These buttons are referred to as *selected*, and show the currently active option. To select a deselected video button, place the mouse pointer inside the video button you want to select, then click the left mouse button. The button will change to reverse video to indicate your choice. A button whose text is displayed in light gray cannot be selected.

Some buttons are mutually exclusive (that is, you can only select one button out of a set). A set of mutually exclusive buttons are called "radio buttons". When you select a new radio button, the previously selected one automatically deselects.

At the bottom of a dialog box there are always two exit buttons, marked OK and Cancel. You must select one of these buttons to exit the dialog box and continue operating STalker. If you select the OK exit button, any changes you made to the dialog box will take effect. If you select Cancel, any changes you made will be ignored. Until you exit the dialog box, operation of STalker is frozen. One of the exit buttons is highlighted with a thick border. This is called the default button. To choose this button without using the mouse, press the [Return] key on the keyboard.

Some dialog boxes have additional exit buttons; these are grouped together with OK and Cancel.

Occasionally, choosing a button will result in the display of a second dialog box. You can recognize this type of button because the text within the button ends in three periods. When you exit this secondary box, you will be returned to the main box. (In STalker, the only exceptions to this rule are the *Settings* buttons in the *Autodialer Directory* Dialog Box. They are too small for the periods to fit.)

In addition to the video buttons, some dialog boxes have text fields. Empty text fields are displayed as a series of underline characters. The thin vertical line in the uppermost text field is the cursor, which shows where your typing will appear. To enter information into the text field where the cursor is located, simply type on the keyboard. Some text fields may expect a certain type of information, such as numbers only. In this case, only the correct keys will be accepted; any others you type will be ignored.

While typing into a text field, you have several options.

Moving: To move the cursor within a field, use the left and right arrow keys on the keyboard.

Deleting: To delete the character to the left of the cursor, press the [Backspace] key. To delete the character to the right of the cursor, press the [Delete] key.

Erasing: To completely erase a field, press the [Esc] key.

To move the cursor from field to field, there are also several options:

1. Move the mouse pointer over the field you want, and left-click.
2. Use the up or down arrow keys on the keyboard.
3. Press the [Tab] key.

The final feature of a STalker dialog box is a vertical scroll bar. The only STalker dialog box in which this appears is the *Autodial Directory* box, which is displayed when you choose the *Autodialer Directory...* option from the **Dial** drop-down menu.

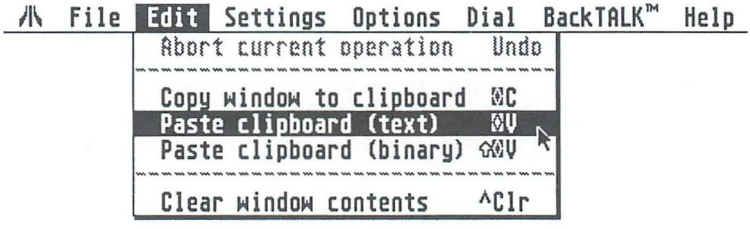
STalker maintains a list of up to 30 numbers in its *Autodialer Directory*, but only 10 are displayed at a time. The scroll bar, displayed to the left of the list of numbers, allows

you to scroll up and down the list. The white rectangle, called the *scroll box*, represents the part of the list you can see, while the gray region represents the part of the list you cannot see. To view different parts of the list, point to the scroll box with the mouse pointer, press and hold the left mouse button, drag the scroll box to a different part of the scroll bar, and release the mouse button. You can also scroll up or down one number at a time by pointing to the up or down arrows at the top or bottom of the scroll bar and clicking the left mouse button.

The *FormDoIt!* utility included in the EXTRAS folder on your STalker disk extends the capabilities of dialog boxes. See the documentation file included with *FormDoIt!* for further information.

5.3. Menu Bar

This is the STalker menu bar, with a menu visible.



5.3.1. Running as a program

When you run STalker as a program, you will see the STalker menu bar at the top of the screen. To activate the menu bar, move the mouse pointer over one of the menu headings. The menu will drop down. To view different menus, move the pointer over the heading of the menu you wish to view.

To make a choice from a menu, move the mouse pointer over the option you want. When the option is highlighted (displayed in reverse video) press the left mouse button. Options displayed in light gray text may not be selected.

Menu options that result in the display of a dialog box always end in three periods.

Some menu options may be toggled on and off by selecting them. The first selection turns the option on, a subsequent selection turns it back off, and so on. When this type of option is toggled on, a check mark will appear in the menu to the left of the option name.

To remove a menu without making a choice, move the mouse pointer completely off the menu, then press the left mouse button.

5.3.2. Running as a desk accessory

When you run STalker as a desk accessory, the stand-alone application currently in use keeps its menu bar at the top of the screen. The STalker menu bar is placed inside the STalker window, directly under the window title bar.

When STalker is running as a desk accessory, its drop-down menus may be accessed only if the STalker window is the currently active (top) window. If it is not, left-click (click the left mouse button) anywhere inside the STalker window.

To activate STalker's menus, place the mouse pointer over the menu heading you wish to view and left-click. The menu will drop down. From this point on, the STalker menus operate in the same manner as explained previously.

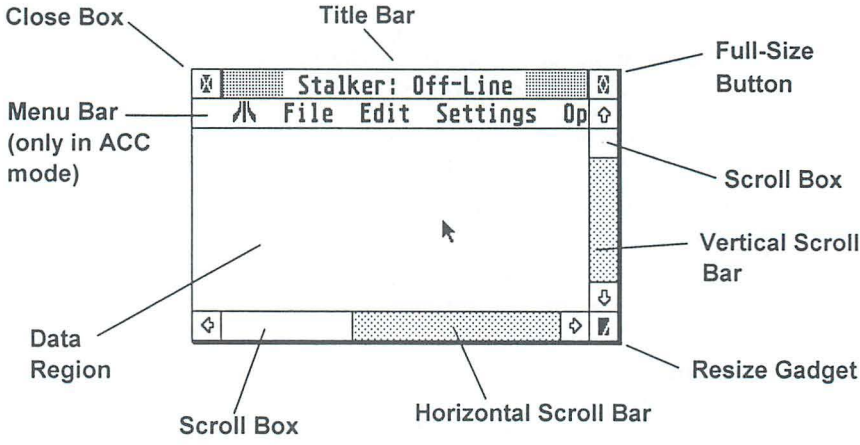
5.3.3. Keyboard shortcuts

When you look at the menu entries, you will notice that some of them have a keyboard shortcut shown to the right of the entry. Typing a keyboard shortcut has the same effect as choosing the menu entry with the mouse. To use a keyboard shortcut, press down the appropriate set of keys, all at the same time. The up arrow character represents the [Shift] key, the exploding box character represents the [Alternate] key, and the caret character (^) represents the [Control] key.

To see a list of additional keyboard commands, press the [Help] key or choose *Keyboard...* from the **Help** menu.

5.4. Window Controls

STalker uses a GEM window which is similar to other GEM windows. However, in addition to the standard GEM method of moving and resizing a window (by using the Drag Bar and Resize Gadget, respectively), STalker also allows you to use the right and left mouse buttons together to move and resize the window. This gives you additional flexibility with the mouse.



5.4.1. Title bar (shows Status Line or Menu Bar)

The top line of the STalker window is called the title bar, and normally displays the current status of STalker (On-Line, Off-Line, File Transfer Complete, etc.) When in Desk Accessory mode, the Menu Bar is located directly below this Status Line. By selecting the Title Bar command on the **Options** menu, you can turn off the Status Line display. If you do this, the Menu Bar will be displayed in the Title Bar. To redisplay the Status Line, choose the Title Bar option a second time.

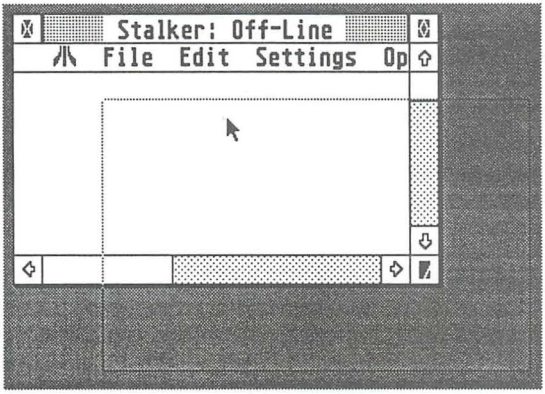
5.4.2. Moving the window

When the Status Line is visible in the Title Bar, you can move the window to a new screen location by pointing anywhere within the Status Line, then pressing and holding the left mouse button. While holding the button, move the mouse and a "ghost window" will move with the pointer. When the "ghost window" is where you want it, release the mouse button and the window will jump to its new location. This operation of holding a mouse button while moving an object to a new location is referred to as "dragging".

When the Menu Bar is visible in the Title Bar, you can drag the window as described above, except that the mouse pointer must point to the blank section of the Menu Bar, to the right of the [Help] entry.

Another way to move the window is to use the right and left mouse buttons together. To do this, first position the mouse pointer anywhere inside the window. Then press and hold down the right mouse button (or the [Alternate] key on the keyboard), and then, while holding the button or key, press and hold the left mouse button. The mouse pointer will change from an arrow to a flat hand. At this point, you can release

the right button or [Alternate] key, but continue to hold the left button. As you move the mouse while holding the button, you will see the "ghost window". To reposition the window, move to the location you desire and release the button.



5.4.3. Close button

The close button, located at the upper left-hand corner of the STalker window, works like any standard GEM window close button.

If STalker is running as a desk accessory, pressing this button will close the STalker window. However, if you are connected to a host system, you will **not** be disconnected, and any data transfer going on will continue to take place. Reopening the STalker window will allow you to check the status of your connection.

If STalker is running as a program, pressing this button will exit the program. If you are in the middle of an upload or download operation, you will not be allowed to exit until the operation is completed or you terminate it.

If you turn off the display of the Status Line by selecting *Title Bar* on the **Options** menu, the close button will also disappear. Pressing [Alternate][Q] will produce the same effect as the close button.

5.4.4. Full Size button

The full-size button, located at the upper right-hand corner of the window, will cause the STalker window to jump between the current window size and the largest possible size (usually the size of the desktop). The first time you click on the full-size button, the window will grow to full size. A second click on the button restores the window to its original size. If you hold down the [Alternate] key while left-clicking, the window will be restored to the size saved in STalker's default information file (the initial window size).

If you turn off the display of the Status Line by selecting *Title Bar* on the **Options** menu, the Full-Size button will disappear along with the Status Line. You can still grow or shrink the window by pointing to the upper right of the Data Region and holding the right mouse button or [Alternate] key while pressing the left mouse button.

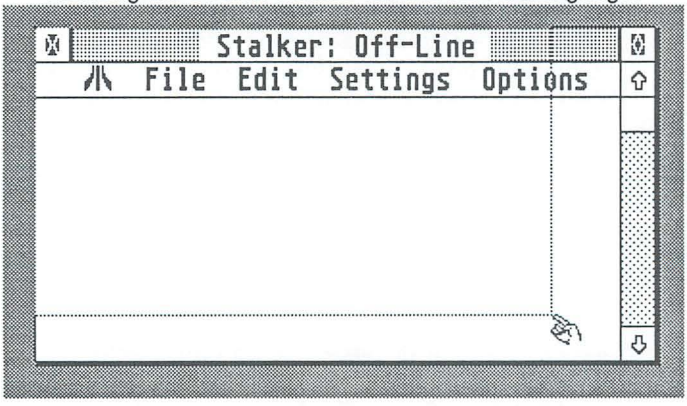
5.4.5. Resize gadget

The Resize Gadget is located in the bottom right-hand corner of the window. You can make the STalker window a new size by using the left mouse button to drag the gadget to the location where you want the window's bottom right corner. Release the button, and the window will resize itself.

Note that if you turn off the Vertical or Horizontal Scroll Bar by using the appropriate entry on the **Options** menu, the Resize Gadget disappears along with the scroll bar.

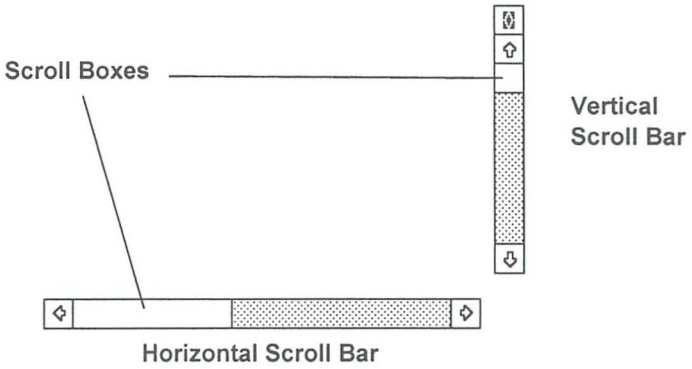
Even with the Resize Gadget missing, you can still resize the window. Simply point the mouse pointer to the bottom right corner of the window (but not within the scroll bar, if one of them is still visible.) Then, press and hold first the right button (or [Alternate] key), then the left button. The mouse pointer will change from an arrow to a pointing hand. At this point, you can release the right button or [Alternate] key, but continue to hold the left button. You can now resize the window. If you get the flat hand (move window) icon rather than the pointing hand, you didn't point to the right place. Release the buttons and try pointing closer to the corner of the window.

Resizing the STalker window without the Resize gadget



5.4.6. Scroll Bars

STalker allows you to have a virtual screen size of up to 9999 lines by 200 columns (virtual screen size is controlled by the *Preference Settings...* command on the **Settings** drop-down menu). This simply means that STalker maintains a buffer (an area of memory) in which it stores up to the last 9999 lines received from the computer system you are connected to. Additionally, the buffer can store more columns than the window can display. The vertical and horizontal scroll bars, at the right hand side and bottom of the window, respectively, allow you to scroll around this virtual screen.



The *vertical scroll bar*, located on the right side of the window, controls which section of the (up to) 9999 line virtual screen is visible. The white rectangle, called the scroll box, represents the part of the virtual screen you can see, while the gray region represents the part of the virtual screen you cannot see.

To go to a different portion of the virtual screen, point to the scroll box with the mouse pointer, press and hold the left mouse button, drag the scroll box to a different part of the scroll bar, and release the mouse button. While dragging, you will see an outline of the bar, called the "scroll drag box".

You can scroll one line at a time by clicking the left mouse button while pointing to the Scroll Arrow at the top or bottom of the scroll bar. This can also be accomplished by holding down the [Control] key and pressing the up and down arrow keys on the keyboard.

You can move up or down a page at a time (the size of a page is the same as the current window size) by clicking the left mouse button while pointing to the gray area above or below the scroll box. This can also be done via the keyboard by holding down the [Shift] key and pressing the up or down arrow keys.

The [Clr/Home] key takes you to the top of the virtual screen, while [Shift][Clr/Home] takes you to the current cursor position.

The *horizontal scroll bar*, located at the bottom of the window, operates in the same manner as the vertical scroll bar, but its keyboard equivalents use the left and right arrows rather than the up and down arrows.

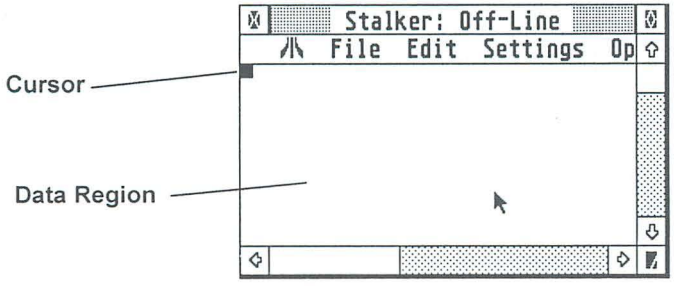
Either or both of the horizontal and vertical scroll bars may be turned off by using the appropriate command in the **Options** menu. This gives you more space on the screen to display data. If you turn off a scroll bar, you can still scroll around the virtual screen by using the keyboard commands ([Shift] arrow to go page by page, arrow keys alone to go line by line, [Clr/Home] to go to the top of the virtual screen, and [Shift][Clr/Home] to go to the current terminal cursor position)

When you turn off one or both of the scroll bars, the resize gadget also disappears. It will reappear if you turn on both scroll bars. Remember that even when the resize gadget is gone, you can still resize the window by using the left and right mouse buttons together, as previously explained.

5.4.7. Data region and virtual screen

The *Data Region* is the main section of the STalker window. This area displays the data sent and received by you and the computer system to which you are connected (called the *host system*). It is your window on the *virtual screen*, in which STalker holds up to 9999 lines by 200 columns of text. Use the scroll bars to scroll around the virtual screen. Use the *Preference Settings...* command on the **Settings** menu to adjust the size of the virtual screen. To erase the virtual screen (not just the part visible in the data region), choose *Clear Window Contents* from the **Edit** menu.

The *cursor*, which is the small black square within the data region, shows where the next character you type, or the next character received from the host system, will appear.



5.4.8. Cut and paste/block functions

Block functions enable you to retransmit, save to disk or print portions of the virtual screen. You can even paste text directly into STeno, Gribnif Software's desk accessory text editor. Once there, you can edit it and then send it back to STalker to be retransmitted!

5.4.9. Selecting a block

"Selecting a block" means that you highlight a piece of text within the virtual screen. The highlighted text is called a block, and you can execute any of the block commands on it.

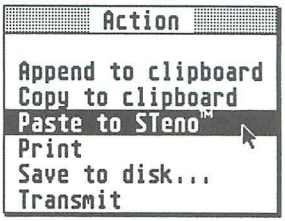
Selecting a block

Stalker: Off-Line
File Edit Settings Options Dial BackTALK™ Help
USERID: MIKEC
PASSWORD:
Welcome to Mike C's Super BBS! Over 25 gigabytes of fun!
Remember: Software Pirates are shot on sight!
Messages: The giant spacewar game tonight starts at 8:00pm sharp! Be sure to log in early for best selection of modem ports!
MAIN LEVEL(0):

To select a block, place the mouse pointer at the beginning or end of the text you wish to select, then press and hold the left mouse button and move the mouse to the opposite end of the desired text. The block will be displayed in reverse video. A shortcut is to hold down the [Shift] key as you press the button; STalker will then select the text between the terminal cursor and the mouse pointer. If the block you want goes off the top or bottom of the data region, just move the mouse past the top or bottom edge of the region (while still holding the button), and it will scroll. If you hold the [Alternate] key, scrolling will be screen by screen. Otherwise it will be line by line. Move the mouse back inside the data region to stop the scrolling. When the block is defined as you want it, release the button, and the *block action window* will appear.

This window has six options which act on your block. Point to an option with the mouse pointer (it will display in reverse video) and left-click. If you choose a block and then decide you do not want to execute a block command, move the mouse pointer completely off the block action window and left-click. The window will disappear and the block will deselect (no longer be selected).

5.4.10. Block Action Window options

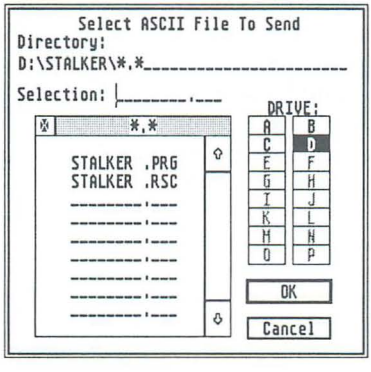


STalker maintains an invisible text clipboard on disk. In addition to the commands in the *Block Action Window*, some of the commands on the Edit drop-down menu also act on this clipboard; see the section of this manual on that menu for further information. If you are using STeno, you should be aware that STalker and STeno normally use the same clipboard file. Therefore, any clipboard operations you do in one program may affect the other as well.

Block Action Window commands are:

- **Append to clipboard :** This command adds the currently selected block to the end of any text already in the clipboard.
- **Copy to clipboard :** When you execute the *Copy to Clipboard* command, any text already in the clipboard is erased and replaced by the currently selected block.
- **Paste to STeno :** If you have STeno in memory, the currently selected block will be copied to STeno's current text file at the current cursor position. It is not necessary to have the STeno window open in order to paste to it.
- **Print :** This option sends the currently selected block to your printer.
- **Save to disk :** This option saves the currently selected block to a text (ASCII) file. You will see the *GEM file selector* (see picture below). Choose the path you want, then enter a filename and choose the OK button. To abort the save, choose the Cancel button. For further information on the GEM file selector, consult your Atari computer owner's manual.
- **Transmit :** The block is sent over the currently selected communications port, using the current text transfer settings. This command is identical to Copy to Clipboard followed by *Paste Clipboard* (*text*) from the Edit menu.

GEM File Selector



6. Introduction to Telecommunications

6.1. Introduction

This section of the manual assumes you know nothing about computer telecommunications. If this is not the case, you may just want to skim this section. It also does not attempt to explain every possible feature of STalker. For detailed information on every STalker command, refer to the *Drop-Down Menu* section of this manual.

6.2. Connecting Computers Together

6.2.1. Modems and null modem cables

Computers communicate over phone lines through the use of a **modem**. A modem, which stands for Modulator-demodulator, transmits by changing information coming out of your computer's serial port into signals which may be transmitted over normal telephone lines. It receives information by taking signals from the phone line (sent by another modem), changing that information back to its original form, and feeding it into your computer's serial port.

Modems typically have three jacks. One is a 25-pin **DB-25 port**. You plug a *modem cable* into this port, and connect the other end to your Atari's serial port. The other two ports are modular phone jacks. One of these jacks (usually labeled "Line") should be connected to a modular wall phone jack via a standard phone cable (usually supplied with the modem). The other jack is connected to your phone using the phone's cable. Thus, when you are not using the modem, your phone works normally. Note that when the modem is connected to another computer, your phone line is in use and anyone calling you will get a busy signal. If you have call waiting, you should turn it off as the call waiting "beep" will interfere with the operation of the modem.

Sometimes you might want to connect two computers which are in the same room by running a cable directly between them. In this case, the modems are not needed as no phone line is involved - the information can be sent directly from one computer to the other. However, you cannot use a regular modem cable for the connection; you must use a **null modem cable**. The reason for this is twofold: the "sexes" of the connectors won't match, and the wiring would be incorrect. A null modem cable is a standard item available at any computer store. As far as the operation of STalker is concerned, it

does not matter whether you are using a null modem cable or a modem. Of course, if you are using a null modem cable, you don't have to dial the other computer as you are already connected!

6.2.2. Hayes and Hayes compatible modems

There are certain commands you can send to a modem, such as telling it to dial a certain phone number, disconnect from a connection, etc. The modem may also be set for various possible options, such as whether or not to answer an incoming call automatically. To simplify communications between computers and their modems, most modem manufacturers have a standardized set of modem commands called the AT set, so-called because all the commands start with AT. Also, modes in the modem are set via a set of built in registers. The command set and registers were created by Hayes, a popular modem manufacturer, and so modems using the AT command set and Hayes registers are said to be "Hayes compatible". You should try to stick to a Hayes or Hayes compatible modem for use with your Atari, as virtually all modem software, including STalker, is preconfigured for this type.

6.2.3. High speed modems

Newer modems automatically adjust to whatever speed the computer talks to them at. Furthermore, these modems will usually try to connect to the remote host at the highest possible rate, by default. For this reason, if your modem's manual says that it can automatically recognize the computer-to-modem connection rate, you should set STalker to always use the fastest communications rate supported by the modem. The modem should decide for itself what rate to connect to the remote modem with.

6.3. Configuration Definitions

Modems have many different parameters, such as the speed at which they transmit, the codes used for each character sent, and so on. Both STalker and the software at the other end of the line must be configured in the same way for communications to proceed properly. To set up the modem, choose *Communication Settings...* from the **Settings** drop-down menu. When you do, you will see the Communication Settings dialog box (see picture on next page) which will enable you to set up your system using the following parameters:

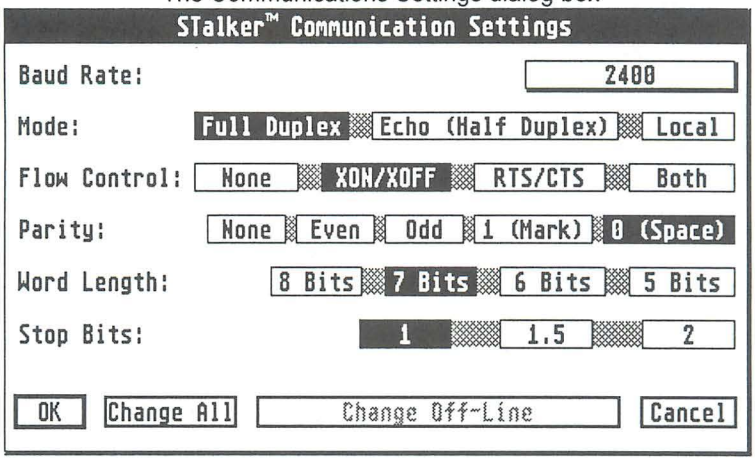
6.3.1. Baud rate

The **Baud Rate** refers to the transmission/reception speed of your modem. Baud Rate is measured in bits per second. The speed at which characters are transmitted is roughly the baud rate divided by 10.

Modems rated at a certain maximum speed, such as 2400 baud, can also use the common lower speeds of 1200 and 300 baud. Therefore, if you have a 2400 baud

modem but you are connected to a computer with a 1200 baud modem, set your modem to 1200 baud.

The Communications Settings dialog box



6.3.2. Full/Half Duplex

Some computers you may connect to will echo back characters you type on your keyboard. This echoing is referred to as full duplex. In full duplex, if you type a 't', that letter will be transmitted to the other computer, and the other computer will then send a 't' back to your computer. Since STalker displays any incoming information from the modem on your screen, the 't' from the other computer will appear. Therefore, if STalker were to also display a 't' at the instant you typed it, you would end up with two 't's! Thus, in full duplex mode, STalker does not display each letter as you type it, but instead waits for the echo.

If the computer does not echo back characters, the connection is referred to as *half duplex*. In half duplex mode, it is necessary for STalker to display each letter as you type it. Here is how to tell which mode to use. If you are connected to a computer and you see two letters appear for each one you type, switch to full duplex mode. If you cannot see anything you are typing, switch to half duplex mode.

STalker also supports a mode called local, in which anything you type appears on the screen but is not sent to the modem.

6.3.3. Word length

Computers communicate through the use of groups of 1's and 0's. Each group represents one character. The *word length* refers to how many 1's and 0's make up one group. The most common word lengths used are 7 or 8. If you connect to a system and

get gibberish on the screen, try changing the word length and parity from 7/even to 8/none or vice versa.

6.3.4. Stop Bits

Stop bits mark the end of a character. Most systems use 1 stop bit.

6.3.5. Parity

Parity checking is a form of error detection. Generally, when you are connected to a system using 7-bit words, you should set parity to Even. When you are connected to a system using 8-bit words, set parity to None.

6.3.6. Flow Control

Sometimes a computer can't process data coming in over a modem fast enough. Flow control permits a computer which is falling behind to tell the other computer to temporarily stop transmitting. When the receiving computer catches up, it can then tell the other end to start transmitting again. XON/XOFF is the most common form of flow control.

At baud rates of 9600 or greater, RTS/CTS flow control should be used. However, not all version of the Atari operating system support this type of flow control correctly. See the *High Speed Modem* section of this manual.

If you are reading information coming in from a host computer and you are falling behind, you can do your own flow control. Just hold down the [Control] key while pressing the [S] key to stop the transmission of data. Press [Control] and [Q] to restart transmission.

6.3.7. Terminal Emulation

STalker emulates three popular terminal protocols: VT-52, VT-100, and PC-ANSI. Which one you will use depends on the system to which you are connecting. Choose your terminal type from the *Terminal Settings* dialog box, viewed by choosing *Terminal Settings...* from the Settings drop-down menu.

PC-ANSI is sometimes called ANSI-BBS, ANSI, et al, by some systems. Also, the VT-52 emulation is the Atari Enhanced VT-52 and contains additional capabilities over and above the standard VT-52.

If you can read text coming in, but the text appears scrambled or has strange characters interspersed with the readable portion, try changing to a different terminal type.

6.4. How to Connect To a host System

Note: The following assumes you have a Hayes-compatible modem.

STalker supports an *autodialer directory* of up to 30 different entries. Each entry will store individual information on the phone number, baud rate, and all the other settings needed. In addition, you can even program the autodialer to log in for you by typing your password and any other necessary information automatically! If you are going to call a BBS or On-Line Service frequently, you should program it into the autodialer directory. If you are only going to call once, you can connect manually. Use of the Autodialer is explained in the section of this manual covering the **Dial** menu. For now, you can just dial manually.

6.4.1. Set configuration, terminal settings, etc.

First, you must set up the communications settings properly using the *Communications Settings* dialog box. If you are dialing into a commercial service like *GEnie* or *CompuServe*, use the settings they give you. If you are dialing a public BBS, generally your settings should be XON/XOFF flow control, and either 8 bits/no parity or 7 bits/even parity. Baud rate and duplex depends on the system you are calling - most BBS's support 2400 baud but not all, and duplex settings vary. Refer to the previous section of this manual for information on these settings.

The next thing you should do is set your terminal emulation to the correct type using the *Terminal Settings* dialog box. If you don't know which type the system you are dialing expects, use VT-52.

Finally, if you are planning to do any file transfer, you may wish to open up the *File Transfer Settings* dialog box by choosing that option from the **Settings** menu and choose the type of transfer protocol you will be using. (Of course, you can also do this after you are connected; see the *File Transfer* section of this manual for further information.)

6.4.2. Dialing

Once your settings are correct, you are ready to dial. Type ATDT (for tone dialing) or ATDP (for pulse dialing) followed by the phone number you want to call. For instance, if you were dialing 1-555-5555, you would enter:

ATDT 1-555-5555 [Return]

to dial. (The dashes are optional.)

If your modem has a speaker, you will hear the process of dialing and connecting as it proceeds. When you are connected, you will be informed by a message on the screen. What happens next depends on the system you are connecting to. Generally, you should press the [Return] key once or twice.

If you get a response but it is completely unreadable, try changing the communication settings from 8 bits/no parity to 7 bits/even parity or vice versa, then press the [Return] key. This usually fixes the problem. If the text is readable but scrambled on the screen, try changing the terminal type.

At this point, you will probably have to enter a password and/or other information; this depends on the system you are calling.

One of the best features of STalker is that when running as a desk accessory, it operates completely in the background. This means that you can use STalker at the same time as another GEM based program. While STalker works on a lengthy file transfer, you can get on with your work - no waiting! If you are running STalker as a desk accessory, once you are connected to a host system you can try this feature out by closing the STalker window and entering a GEM based program, then reopening the STalker window. Try using STalker and the other program at the same time.

6.4.3. Hanging up

When you are done communicating, log off from the host system (the system you are connected to). Then choose *Hang Up Phone Line* from the **Dial** menu. Last of all, you should turn off your modem, as if you leave it on, it may answer the phone when you don't want it to!

6.5. File Transfer

One of the major reasons to call other computers is to transfer programs or files between your computer and the other computer. Sending a file is called *uploading*, while receiving a file is called *downloading*.

Remember that STalker uploads and downloads in the background. This means that while you transfer a long file to or from another computer, you can run another GEM based program and get on with your work. Once the transfer commences, you can even close the STalker window. STalker will notify you when the file transfer is complete by playing a short tune through your computer's speaker.

6.6. Basics of Uploading/Downloading

6.6.1. Text (ASCII) Files

The simplest way to perform a download or upload is to send the information character by character from the first computer to the second. In fact, when you receive some text from the host system and then use STalker's cut-and-paste function (*Save to Disk* on the *Block Action Window*) to save some text in the virtual screen to disk, you have performed an elementary download operation. If you use the *Transmit* option from the *Block Action Window*, you have performed an upload operation.

In addition to these simple upload/download methods, which operate only on text in the virtual screen, STalker also allows you to send a text file stored on disk to the host computer, or receive some text and immediately store it to disk.

- Text File Upload - To send a text file, use the *Send Text File...* command on the File menu. Upon issuing the command, you will see the GEM file selector. Choose the file you wish to send, and it will be transmitted to the host computer. Because this is a text upload, the host computer receives the file one character at a time, just as if you had typed it yourself. (Of course, you'd have to be a pretty fast typist!) Watch the status line at the top of the STalker window for information on the status of the upload.

  This is a useful function for electronic mail. You can use your favorite word processor to compose your letter, then save it as an ASCII (text) file and upload it later. Gribnif Software's STeno text editing program is especially convenient for this, as you can run both STeno and STalker as desk accessories.

- Text File Download - To receive some text and immediately save it to disk, choose *Receive Text File...* from the File menu. The GEM file selector box will pop up to allow you to name the file in which your Atari will store the data. As soon as you choose the OK button to exit the file selector, STalker will start saving incoming text. At this point, you should type the appropriate command to your host system to start sending the text you wish to receive. When you are finished receiving, choose *Receive Text File...* again to complete the transfer. Watch the status line at the top of the STalker window for information on the status of the download.

  Note that at any time while incoming text is being saved, you can still type on the keyboard to send instructions to the host computer.

6.6.2. Binary Files

Sending and receiving files as straight text works fine, but there are problems. Sometimes there might be noise on the phone line or other problems which can garble the data. Since there is no way to tell what the data is supposed to look like, there is no way to check for errors. Also, some files (such as programs) are not readable text and cannot be easily transmitted as text.

To bypass these problems, use *binary file transfer*. With this method, the data is transmitted along with additional information which allows the receiving computer to verify that the data arrived correctly. If it did not, the receiving computer can request the host computer to re-transmit the data. STalker supports four basic methods, called protocols, of binary file transfer, plus some variations (these are explained later in this section). These are *XModem*, *YModem*, *CompuServe B+*, and *ZModem*. To choose the protocol you want, use the File Transfer Settings... command from the **Settings** menu. The *File Transfer Settings* dialog box will appear, (see picture on page after next), and you can select the protocol you want by left-clicking on the button next to the word

"Protocol", then making your choice. Exit the dialog box by left-clicking on the OK button.

- **Binary Uploads** - If you want to do a binary upload of information from the virtual screen to the host computer, first store the information you want in the clipboard (see Cut and *Paste/Block Action Window* elsewhere in this manual.) Next, choose the transfer protocol you want, if you have not already done so. Then, tell the host computer to get ready for a binary upload (exactly how this is done depends on the system to which you are connected). Finally, choose *Paste Clipboard (Binary)* from the **Edit** menu. The transfer will begin and you will see a status dialog box. To abort the transfer before completion, choose the **Abort Transfer** button at the bottom of the dialog box or press the [Undo] key.

  If the information you want to upload is stored in a disk file, you first need to choose the transfer protocol you want, if you have not already done so. Next, tell the host computer to get ready for a binary upload (exactly how this is done depends on the system to which you are connected). Finally, choose *Upload Binary File...* from the **File** menu. The GEM file selector will appear so you can choose the file to upload. After you exit the file selector, a status dialog box will appear with information on the progress of the upload. To abort the transfer before completion, choose the **Abort Transfer** button at the bottom of the dialog box or press the [Undo] key.

  If the host you are using supports the ZModem file tranfer protocol, and STalker is set to "Auto-initiate ZModem file transfer" (in the *File Transfer Settings* dialog), you do not need to choose **Upload Binary File**. Instead, you will be presented with the GEM file selector automatically, as soon as the host computer starts to download.

- **Binary Downloads** - When doing a binary file download, it is sometimes unnecessary to use the *File Transfer Settings...* dialog box to choose a protocol because STalker can automatically detect which protocol the host computer is using. However, this does not work for YModem-G. If you are using YModem-G, you will need to choose that protocol, and if not, choose one of the other protocols.

  To receive a file, tell the host computer to get ready to start a binary download (exactly how this is done depends on the system to which you are connected). Then, select *Download Binary File...* from the **File** menu. The GEM file selector will appear so you can name the file to be downloaded. After you exit the file selector, a status dialog box will appear with information on the progress of the download. To abort the transfer before completion, choose the **Abort Transfer** button at the bottom of the dialog box or press the [Undo] key.

  If the host you are using supports the ZModem file transfer protocol, and STalker is set to "Auto-initiate ZModem file transfer" (in the *File Transfer Settings* dialog), you do not need to choose **Download Binary File**. Instead, you will be presented

with the GEM file selector automatically, as soon as the host computer starts to download.

- If you type [Shift][Alternate][D], STalker will immediately begin a binary download without prompting you for a filename. The file will be named RECEIVED.FIL unless you are using YModem or ZModem, which transmit the filename.

6.6.3. Binary file transfer protocols

- **XModem** - XModem is the most common binary file transfer protocol. In fact, almost all systems support it. There are two versions - standard XModem with 128 byte blocks, and XModem with 1K blocks (both checksum and CRC variants are supported automatically). If your host system supports it, use the faster 1K block protocol. If your host system just says "XModem" without specifying, use the 128 byte block protocol. When receiving, STalker can tell them apart automatically.

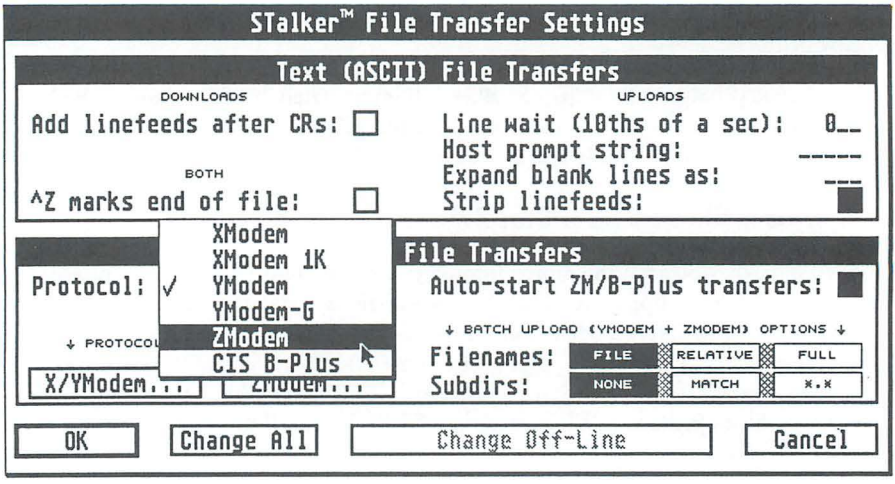
- **YModem** - STalker supports two versions of the YModem protocol: YModem and YModem-G. Use YModem-G only with a null modem cable or error-correcting modem, as it does not do error-correction! If an error occurs, the entire transfer is canceled. However, the lack of error correction makes YModem-G faster than regular YModem or even ZModem.

- **ZModem** - ZModem uploads and downloads are automatically detected. Not only do you not need to change the protocol choice setting in the *File Transfer Settings...* dialog box to ZModem before doing a ZModem download, but STalker will automatically begin the download without your issuing the *Download Binary File...* command. Similarly, if the host computer requests a ZModem upload, your computer will comply without your issuing an upload command.

  You can turn off this automatic ZModem detection by opening the File Transfer Settings dialog box from the **Settings** menu and deselecting the button to the right of the words "auto-initiate ZModem/B-Plus transfers".

- **B-Plus** - This file transfer protocol is used exclusively with CompuServe Information Services. In order to use this protocol, you must either be using STalker as a program, or you must at least be using MultiTOS or the Geneva multitasking environment. Also, the BackTALK script directory must be set to the location of BPLUS.TTP.

A major difference between XModem and Y/ZModem is that XModem does not transmit the filename of the file you are uploading or downloading - it only transmits the data. YModem and ZModem transmit the filename and time of the transfer along with the data. For this reason, YModem and ZModem let you send more than one file in sequence. Doing this is called a *batch upload*; see the section of this manual describing the *Upload Binary File...* menu option for details.



Note: If additional File Transfer Protocols have been added since the printing of this manual, they will appear in the above dialog box, and be documented in the READ_ME.NOW file on the disk.

7. Drop-Down Menus and Dialog Boxes

This chapter explains all of STalker's drop-down menu commands and any resulting dialog boxes in detail. To save space, the word "Alt" is used instead of the word "Alternate". For example, "[Alt][I]" means you should hold down the [Alternate] key while pressing the [I] key.

7.1. Desk (or Atari Fuji) Drop-Down Menu

7.1.1. STalker info... [Alt][I]

Selecting this option displays a dialog box with important information about the version of STalker being used. Write down the version number and serial number displayed in this box before calling Gribnif Software for technical support.

This dialog box also displays how much free RAM you have.

To exit the box, click the left mouse button or press any key.

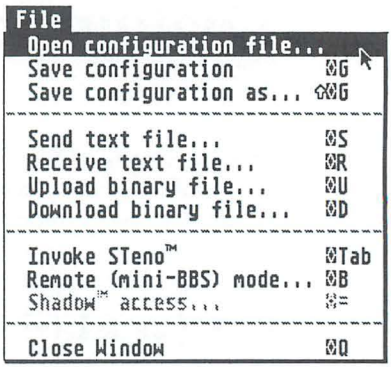
7.1.2. Desk accessories

If you are running STalker as a stand-alone program, any desk accessories you have loaded will appear on this menu and may be accessed normally.

If you are running STalker as an accessory, the only entry on the menu will be *STalker Info.*

7.2. File Drop-Down Menu

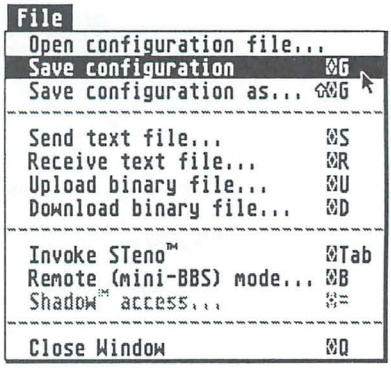
7.2.1. Open configuration file...



STalker allows you to have multiple configuration files saved on disk. These files hold all the setup information from the various dialog boxes, such as the baud rate, etc.

When you choose this option, you will see the GEM file selector. Choose a configuration file, select the OK button, and the file will load. Choose Cancel to abort the command. Values in the *Preference Settings* dialog box that require a reboot to change will not be affected by loading a new configuration file.

7.2.2. Save configuration [Alt][G]

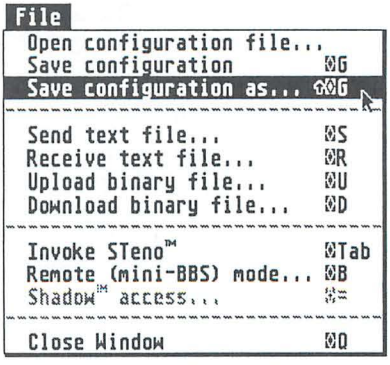


This option lets you save the current configuration. The filename used will be the last name given in an *Open Configuration File...* command. If you have not executed this command, STALKER.INF is the default filename.

The STALKER.INF file always loads automatically each time you load STalker. If you want to use a different file, use *Open Configuration File...* to load the file you want.

Note: This option saves all the autodialer settings. It is therefore not necessary to save these settings separately in the autodialer in order for the settings to be loaded automatically when STalker first runs.

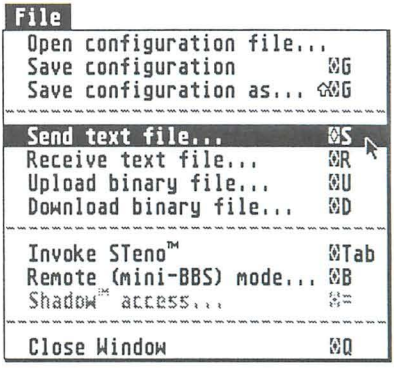
7.2.3. Save configuration as... [Shift][Alt][G]



When you issue this command, you will see the GEM file selector. Choose the path and filename you want to save your current configuration to, then choose the OK button to save the data. Choose the Cancel button to cancel the command.

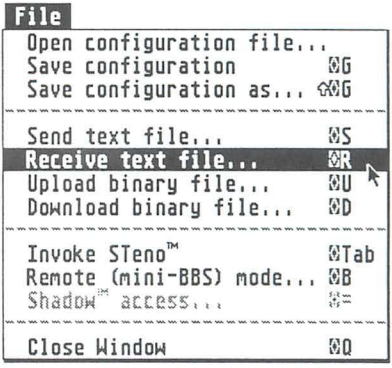
7.2.4. Send text file... [Alt][S]

This command enables you to send a text (ASCII) file to the host computer. Upon issuing this command, you will see the GEM file selector box. Choose the file you wish to send, and it will be transmitted to the host computer using the current Text Transfer settings in the *File Transfer Settings* dialog box. Watch the status line at the top of the STalker window for information on the status of the upload. To stop the transfer in progress, press the [Undo] key.



7.2.5. Receive text file... [Alt][R]

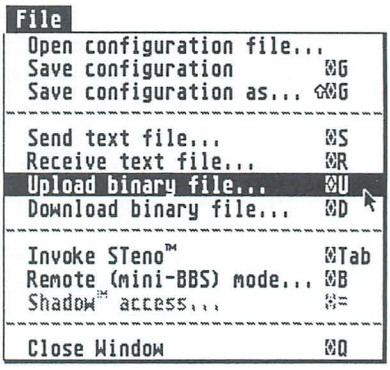
This command lets you save incoming text to disk. When you issue this command, the GEM file selector will appear to allow you to name the file in which your Atari will store the data. As soon as you choose the OK button to exit the file selector, STalker will start saving incoming text. At this point, you should type the appropriate command to your host system to start sending the text you wish to receive. When you are finished receiving, choose *Receive Text File...* again to complete the transfer. Watch the status line at the top of the STalker window for information on the status of the download.



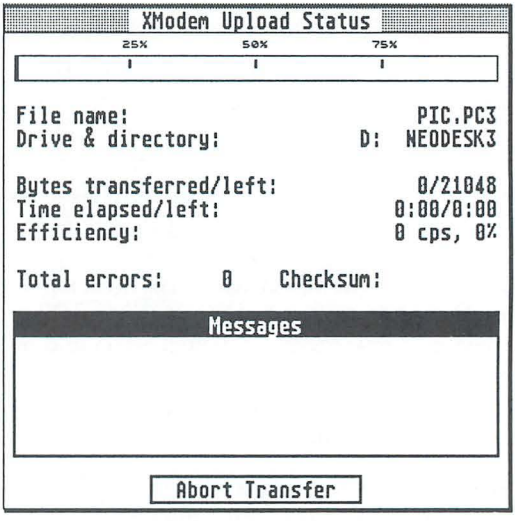
Note: At any time while incoming text is being saved, you can still type on the keyboard to send instructions to the host computer.

7.2.6. Upload binary file... [Alt][U]

This command lets you send a file to the host computer, using the settings and binary file transfer protocol currently selected in the *File Transfer Settings...* dialog box. Before selecting this command, tell the host computer to get ready for a binary upload (exactly how this is done depends on the system to which you are connected). Then, choose the command. The GEM file selector will appear so you can choose the file to upload. After you exit the file selector, a status dialog box will appear with descriptive information on the progress of the upload.



Status messages and errors are displayed in the *Messages* portion of the box. Efficiency rates the speed of your data transfer where 100% is the theoretical maximum speed (baud rate divided by 10). To abort the transfer before completion, choose the *Abort Transfer* button at the bottom of the dialog box or press the [Undo] key.

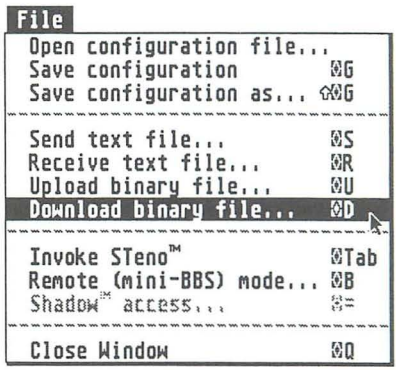


When you are using YModem or ZModem, you can do batch uploads. Batch uploads let you send a group of files with one upload command. To do this, enter a file specification using wildcards. Wildcards are special characters that substitute for all or part of a filename. The '*' character substitutes for a whole section of a filename, while the '?' character substitutes for only one letter.

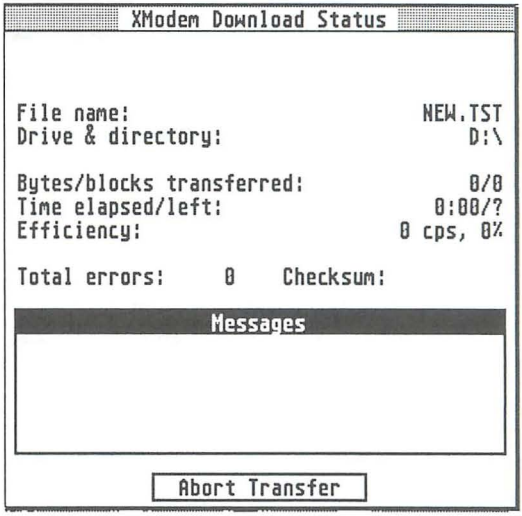
For instance, if you wanted to send all files with a '.TXT' extension in a certain directory, you would first choose that directory in the GEM File Selector, and then enter *.TXT for a filename. If you wanted to send all files called STUFF, regardless of the extender, you would enter STUFF.* for a filename. If you had a series of files called STUFF1.TXT, STUFF2.TXT, and STUFF3.TXT, you would enter STUFF?.TXT for a filename. Parameters for batch uploads are set by invoking the *File Transfer Settings...* command on the Settings drop-down menu. Refer to the instructions for this command for further details.

7.2.7. Download binary file... [Alt][D]

This command lets you receive a file from the host computer. STalker will try to automatically detect the binary transfer protocol your host system is using, so it is not necessary to access the *File Transfer Settings...* dialog box unless, you are using YModem-G as your transfer protocol.



First, tell the host computer to get ready for a binary download (exactly how this is done depends on the system to which you are connected.) Then, choose the command. The GEM file selector will appear so you can name the file to be downloaded. After you exit the file selector, a status dialog box very similiar to the *Upload Status* dialog box will appear with descriptive information on the progress of the download. Status and error messages are displayed in the *Messages* portion of the dialog box. To abort the transfer before completion, choose the Abort Transfer button at the bottom of the dialog box or press the [Undo] key. When the transfer terminates, you will hear a short tune, and a message will appear in the STalker window.

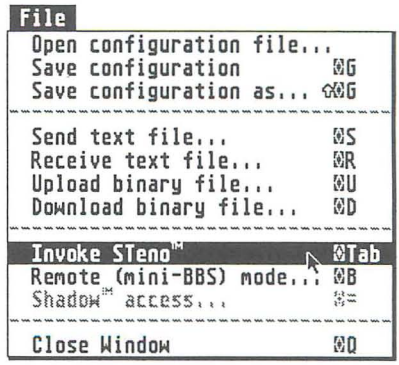


7.2.8. Invoke STeno [Alt][Tab]

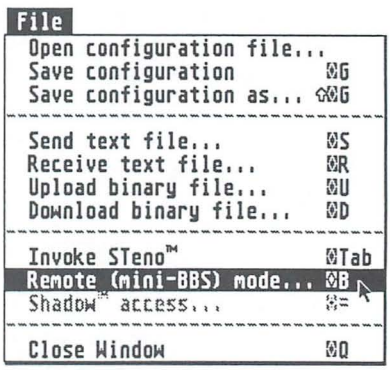
If you have STeno, Gribnif Software's text editor loaded, this command will open and activate the STeno window.

If STeno is not loaded, this option will be grayed out and will not be selectable.

See the chapter *Using STeno with STalker* for information on setting up STeno.

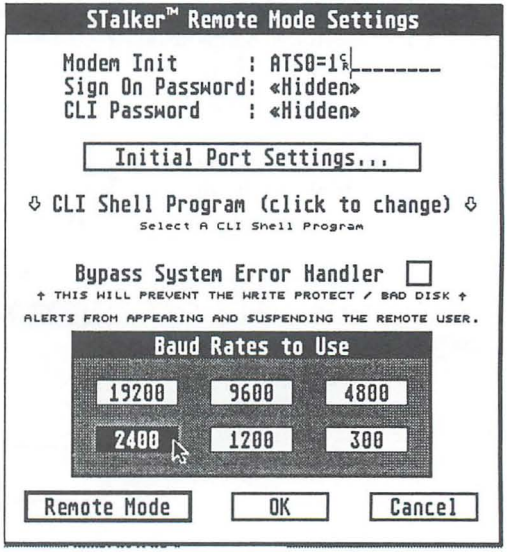


7.2.9. Remote (mini-BBS) mode... [Alt][B]



This option allows you to dial in with another modem-equipped computer and access your system. This function is useful if you will be travelling and may need to upload or download files from your system.

When you issue this command, you will see the *STalker Remote Mode Settings* dialog box. By configuring the dialog box as described below, then exiting the box with the Remote Mode exit button, you can access your system via another computer/modem. The computer you call from is called the remote computer.



At the top of the *Remote Mode Settings* dialog box, there are three text fields labelled Modem Init, Sign On Password, and CLI Password. The default value for Modem Init is correct for Hayes compatible modems. If your computer uses a different type of modem, change this field to the appropriate value for that modem. The Sign On Password field is the password you will need to enter when you connect to your system. Clear the field by pressing the [Esc] key, then type in your desired password. The CLI Password field lets you enter a password to restrict access to a CLI (Command Line Interpreter) program which may be accessed from the remote computer. Enter a password here. When you exit the Remote Mode Settings dialog box, these passwords will be hidden, and you will not be able to view them again. If you forget your entries, you must clear the fields and enter new passwords. If either password is left completely blank, STalker will not ask for a password before granting access to the caller.

The Initial Port Settings... button lets you configure the setup of your modem. The resulting dialog box is the standard STalker communications settings box, described later in this chapter. It is helpful if you remember what settings you use here, as when you dial in you may need to configure the remote computer accordingly. If you select Half Duplex mode, someone looking at your computer screen will see everything the

remote user sees except when the CLI is in use. If you select Full Duplex, STalker will not show anything in the terminal window except modem commands and responses.

Click the mouse on the words CLI Shell Program (click to change) and the GEM file selector will appear. Choose a TOS based CLI program and its name will appear in the dialog. This CLI program can be run if you call into your system and know the CLI password. Only CLI programs that support redirection will work, such as the public domain CLI on the STalker disk.

Note: If you are accidently disconnected while using the CLI, you will not be able to log back in.

If you left-click in the box next to the words Bypass System Error Handler, you can turn off the dialog boxes that normally pop up on your Atari's screen when you have a bad disk, write protect error, etc. As these boxes freeze the system until the problem is corrected, and you would be some distance away at the remote computer, this can save you from being locked out of your system in the event of an error.

The Baud Rates to Use buttons allow you to choose which baud rates you can use when calling in. You must choose at least one speed before you can enter remote mode.

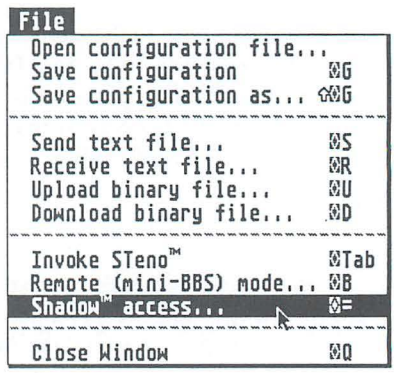
If you exit the *Remote Mode Settings* box with the OK button, changes you made to the box will take effect, but the system will not enter remote mode. If you exit via the Cancel button, any changes you made will be ignored and the box will return to its previous state. If you exit via the Remote Mode button, the system will enter remote mode and start waiting for an incoming call. Before you leave, be sure your modem is turned on. To exit remote mode, press the [Undo] key.

When you dial in, you will be greeted with "Hit Return" messages until you type [Return]. (Wait for this, as STalker cycles through the baud rates.) After entering the password, you will see the prompt "Command?". Type "?" for a list of commands. Commands are available to choose a directory, list the files in it, upload or download files, run the CLI program, and log off. There is also a Chat Mode, in which you and a person at your computer can type messages to each other.

Remote mode is actually run by a BackTALK script called REMOTE.BTK, located in the script directory (see *Preference Settings* for details.) Therefore, the script directory must be set properly in order for remote mode to work. If you want to try out Remote Mode without actually dialing in, use Communication Settings... on the Settings menu to set your mode to 'Local', then execute the REMOTE.BTK script by using *Run BackTALK Script...* from the BackTALK menu. If you are told your script buffer is not big enough, use *Preference Settings* to increase its size. You will have to save your settings and reboot to reload the accessory, or reload the STalker program, for the change to take effect.

7.2.10. Shadow access... [Alt][=]

If you own the *Shadow* background file transfer utility from Double Click Software, you can use STalker to interface with the Shadow driver, rather than using the desk accessory supplied on the Shadow disk. You can still use STalker's file transfer routines if you wish; it's your choice.

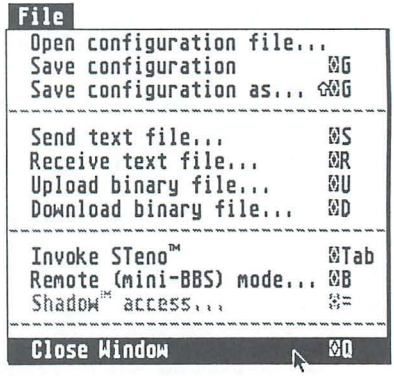


If Shadow is loaded, the *Shadow Access...* command will be enabled. This will present you with one of two dialog boxes similar to the ones used by the Shadow Desk accessory. The first box allows you to start a Shadow upload / download, and the second allows you to monitor the status of a transfer and abort it if necessary.

7.2.11. Quit/Close window [Alt][Q]

As a GEM program

When run as a GEM program, this option will exit STalker and return you to the desktop. It will only say Quit. If you are in the middle of a file transfer operation, you will not be allowed to exit until the transfer finishes or is aborted.

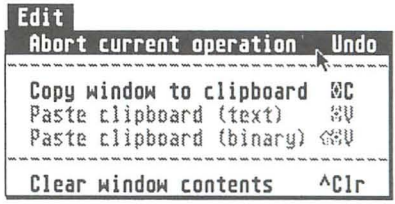


As a desk accessory

When run as a desk accessory, this option closes the STalker window. It will only say Close window. As STalker operates in the background when used as a desk accessory, the only thing this command does is remove the STalker window from the screen. You are not disconnected from the host system, and any data transfer operation in progress will continue. To reopen the STalker window, choose STalker from the Desk or Atari Fuji menu.

7.3. Edit Drop-Down Menu

7.3.1. Abort current operation [Undo]



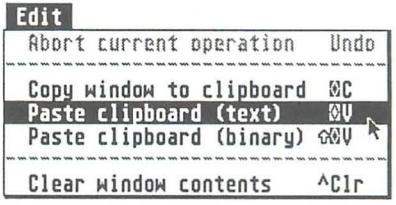
This command lets you stop any operation in progress. This includes file transfer operations and remote mode. You can also press the [Undo] key on the keyboard.

Some operations, like an autologin sequence, actually consist of multiple steps. Pressing [Undo] will cancel just the current step. To abort the entire operation press [Shift][Undo].

7.3.2. Copy window to clipboard [Alt][C]

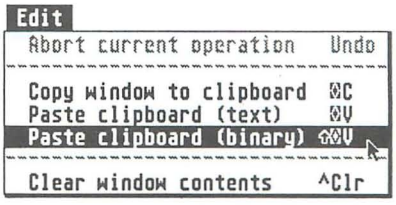
This command copies the visible contents of the window (not the entire virtual screen) to STalker's invisible text clipboard, erasing any text already in the clipboard. If you wish to append to text already in the clipboard, define a block and use the *Append to Clipboard* command from the *Block Action Window*.

7.3.3. Paste clipboard (text) [Alt][V]



This command uploads the contents of the clipboard to the host system as text. It is similar to the *Send Text File...* command on the File menu, but sends text from the clipboard rather than from a disk file.

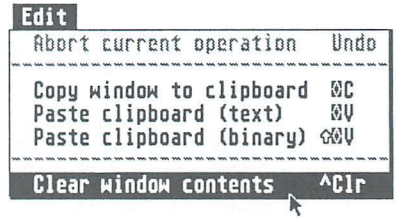
7.3.4. Paste clipboard (binary) [Alt][Shift][V]



This command uploads the contents of the clipboard to the host system, using the *Binary File Transfer Protocol* and settings currently in effect. Before issuing this command, you should tell your host system to prepare for a binary upload.

7.3.5. Clear window contents [Control][Clr/Home]

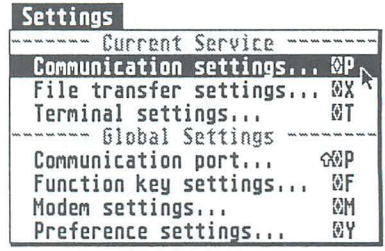
This command erases the entire virtual screen, not just the text that is visible within the window.

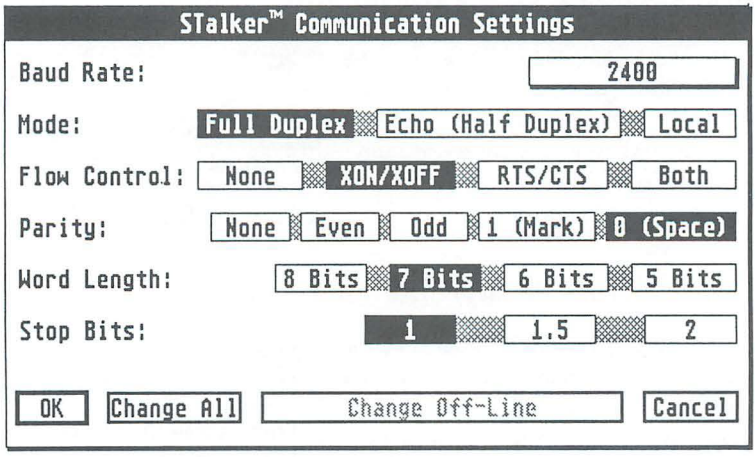


7.4. Settings Drop-Down Menu

7.4.1. Communications settings... [Alt][P]

This option displays the *Communication Settings* Dialog Box. You can set the baud rate, echo mode (duplex), flow control, parity, word length, and stop bits by left-clicking on the appropriate set of radio buttons. See the *Intro to Telecommunications* chapter for information on these settings.



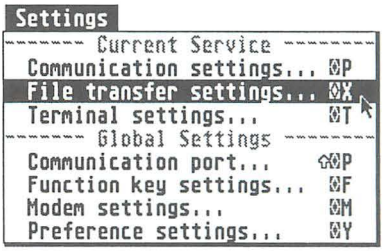


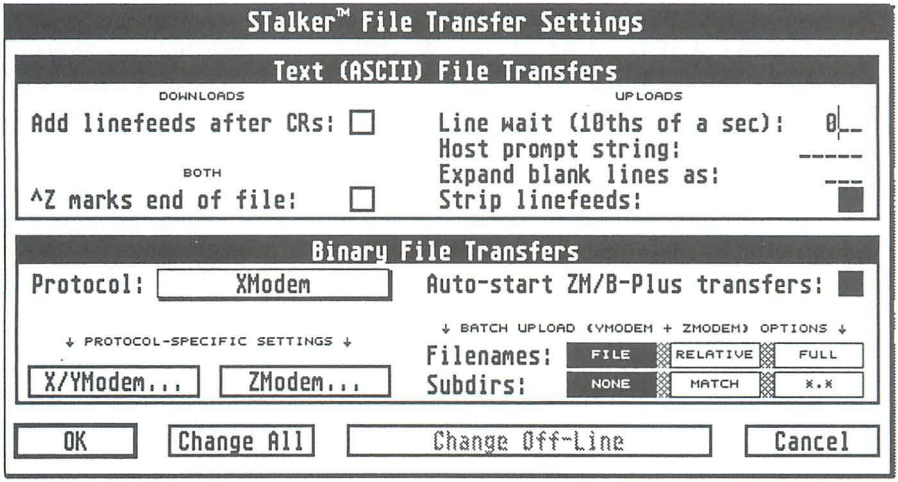
There are 4 ways to exit from this dialog box. The OK button puts the changes you made to the box into effect until you change them again or dial using the Autodialer menu. The Change All button puts the changes into effect and copies them to all entries in the Autodialer. (Each of the 30 entries in the autodialer not only stores a phone

number, but a complete and independent set of parameters; see *Autodialer Directory* for more information.) If you have used the Autodialer to call a system, the Change (name) button changes only the Autodialer entry you are using to reflect the new parameters. Finally, the Cancel button exits the dialog box and ignores all changes.

7.4.2. File transfer settings... [Alt][X]

The *File Transfer Settings* dialog box, displayed by invoking this command, lets you set the parameters for both text and binary uploads and downloads. The top half of the box deals with text uploads and downloads, while the bottom half deals with binary file protocols.





There are 4 ways to exit from this dialog box. The OK button puts the changes you made to the box into effect until you change them again or dial with the Autodialer menu. The Change All button puts the changes into effect and copies them to all entries in the Autodialer. If you have used the Autodialer to call a system, the Change (name) button changes only the Autodialer entry you are using to reflect the new settings. The Cancel button exits the dialog box and ignores all changes.

7.4.2.1. Text settings

The button labelled Add linefeeds after CRs, when turned on, causes all Carriage Return characters received in a text download to have Linefeed characters added after them before the file is saved. This is useful because many text editing programs expect

each line in a text file to end with a CR/Linefeed combination. Most systems already send the CR/Linefeed combination, so this option can usually be turned off.

The ^Z (Control Z) marks end of file option, when turned on, causes STalker to add a ^Z marker to the end of the newly saved text file when the *Receive Text File...* command is terminated. In *Send Text File...* with this option on, STalker uses ^Z as an end-of-file marker and will therefore stop transmitting when it encounters a ^Z. With this option off, STalker transmits until the physical end of the file.

The Line wait text field lets you enter a number which is used to pause uploads at the end of each line. The number is in 10ths of a second, so a 10 in this field would cause a pause of one second between transmission of each line. Use this function if your host computer needs extra processing time before it is ready to receive the next line.

Host prompt string is dependent on your host system; check with your host system to see what, if anything, to enter in this field. When sending a text file, STalker will wait up to 30 seconds for the specified Host Prompt String to be received before sending the next line of text. If it appears that the Host Prompt String is not coming, you can abort the wait by pressing the [Undo] key. [Shift][Undo] or *Abort Current Operation* from the Edit menu aborts the entire file transfer command. If you have a Line Wait as well as a Host Prompt String, STalker will first wait for the string and then pause for the line wait.

The Expand blank lines as text field lets you enter characters which will be transmitted in place of any blank lines in your upload file. This is useful because many BBS text editors exit the editor upon receipt of a blank line. If you are trying to send text containing blank lines into an editor of this type, you can use the Expand Blank Lines field to substitute a space for each blank line. This will enable you to send your file without any problems.

Finally, the Strip linefeeds option will remove linefeeds from your file's transmission. Carriage Return/Linefeed combinations will therefore be transmitted as Carriage Returns alone.

7.4.2.2. Binary File protocol settings

The Protocol button lets you select which Binary File Protocol you will use for future uploads. Left-click on the button to pop up the protocol menu, then left-click on the protocol you want.

Note: If any new protocols have been added to STalker since this manual was written, they will be added to the protocol buttons in this dialog box. See the README.NOW file on the program disk for information, if necessary.

When downloading, STalker can usually detect which protocol the host computer is using. However, if STalker is set for YModem-G it will not be able to detect any other protocol.

The X/YModem... and ZModem... buttons pop up dialog boxes which enable you to set up parameters for X,Y, and ZModem. Generally, the default settings work. If they don't, check with the system you are calling to see what the correct settings are. These boxes are explained on the next page.

YModem and ZModem can be used for batch uploads, as explained in the *Binary Upload...* menu option. Because they can transmit several files in sequence, YModem and ZModem send the filename along with the file. The Filenames group of radio buttons lets you control how the filename is sent. File sends just the filename with no subdirectory information. Relative adds subdirectories below the starting subdirectory, and Full sends the complete pathname (except the drive letter, which is never sent). For example, if your upload request path was for C:\TELECOMM*.* and the current file being uploaded was C:\TELECOMM\DIR\INFO.TXT, you would send the following:

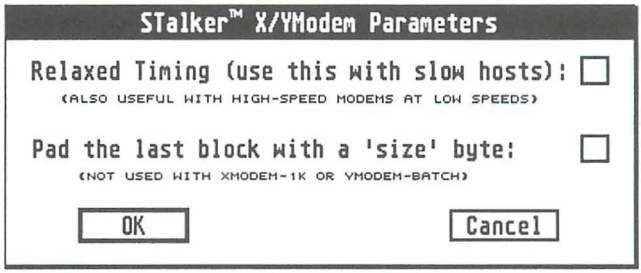
```
FILE:      info.txt
RELATIVE:  dir/info.txt
FULL:      telecomm/dir/info.txt
```

The *Subdirs* group of radio buttons lets you control how subdirectories are handled when doing a batch upload. None means do not look in subdirectories; rather, only send files in the requested directory. Match means to look in all subdirectories for files that match the requested template. '*.*' means to send all files in all subdirectories.

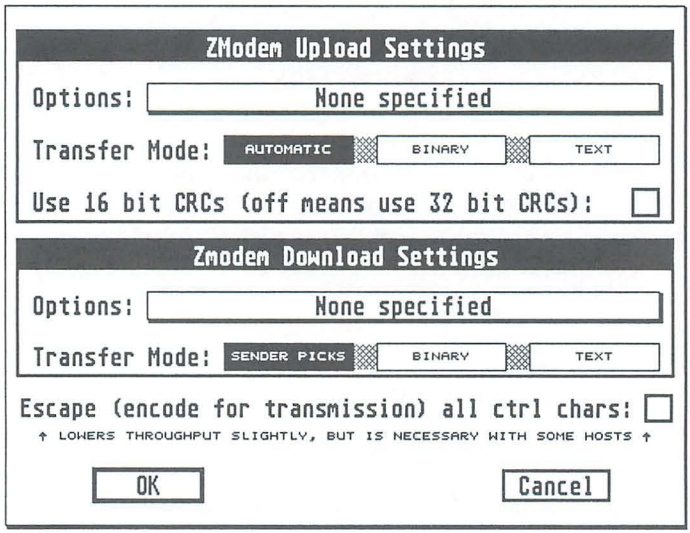
The Auto-start ZM/B-Plus transfers button controls whether or not STalker automatically begins a ZModem upload/download upon receipt of the special string indicating a ZModem up/download (**^XBO). A similar string is also detected for CompuServe B-Plus file transfers.

7.4.2.3. Setting X/Y/ZModem Upload/Download Parameters

In the *X/YModem Parameters* box, click on the appropriate button to turn the two options on or off. Normally, both the options may be off.


STalker™ X/YModem Parameters
Relaxed Timing (use this with slow hosts):
(ALSO USEFUL WITH HIGH-SPEED MODEMS AT LOW SPEEDS)
Pad the last block with a 'size' byte:
(NOT USED WITH XMODEM-1K OR YMODEM-BATCH)
OK
Cancel

The *ZModem Settings* box contains a number of buttons which let you set the parameters for ZModem uploads and downloads.



The two Options buttons pop up a menu of choices as follows:

None specified: Use whatever option, if any, the host system specifies. Unless you choose this option, STalker will override the host system's specified option with its own.

Transfer if source is newer or longer: send the file only if the destination file does not exist, or has a timestamp or length less than the source file.

Transfer if size/date/content differs: send the file only if the destination file does not exist, or has a timestamp or length different than the source file. If the timestamp or length matches, the receiver should request that the sender compute a 32-bit CRC on the files to see if they are the same. If they are not, the file is transmitted. Note that many ZModem implementations do not support the CRC section of this option.

Append: Append the contents of the source file to the destination file, creating the destination file if it does not exist.

Overwrite: Delete the destination file, then transfer the file.

Transmit only if source newer: Send the file only if the destination file either does not exist or has a timestamp 'older' than the source file.

Don't overwrite: Do not transfer the file if a file of the same name already exists on the destination machine.

The Transfer Mode button groups control whether ZModem text or Binary is used. In ZModem binary mode, the file is sent exactly as it is, byte for byte. In ZModem text mode, the sender does whatever translation is required to change end-of-line markers within files to be transferred between computers that use different methods for marking the end of a line. Files to be transferred via ZModem Text must contain only printable ASCII characters (from ASCII 33 to 126, plus spaces, tabs, backspaces, LF's and CR's).

You can usually leave the Transfer Mode buttons set to *Automatic* and *Sender Picks*. STalker can usually pick the correct mode so it is not necessary to specify which one to use under normal circumstances.

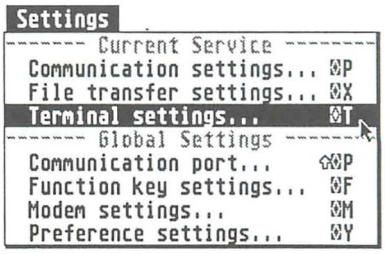
If STalker is receiving in ZModem Text mode, it will convert single Linefeeds into CR/LF pairs. Due to the fact that the file is being modified as it is received, files sent in Text Mode cannot have an aborted transmission resumed.

The Use 16 bit CRC's button lets you choose between 16 and 32 bit CRC's. When on, you will get 16 bit CRC's, when off, 32 bit. You should generally leave this button off, as 32 bit CRC's are more reliable.

The Escape All Control Characters button causes all 7 and 8 bit control characters to be 'escaped' (preceeded with an Escape sequence). This slows down transfer speed but is sometimes the only way to transfer binary files over systems that try to process control characters.

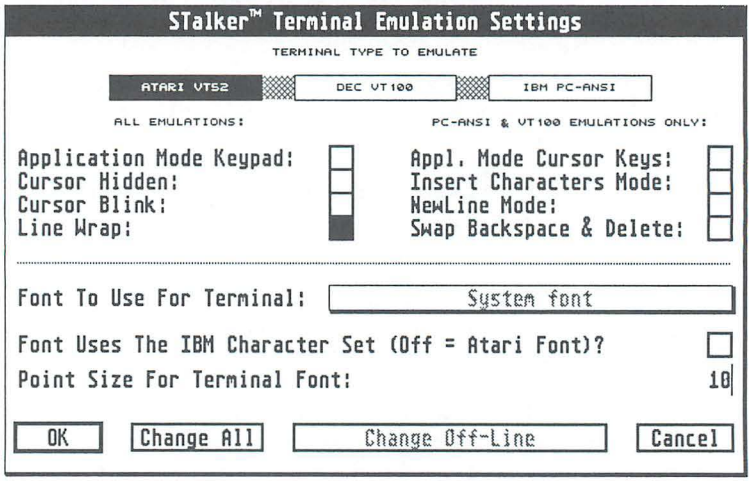
7.4.3. Terminal settings... [Alt][T]

The *Terminal Settings* dialog box, which appears when you invoke this command, lets you choose which of STalker's three terminal emulation modes you wish to use - VT52, VT100, or PC-ANSI. You can also set various parameters related to the emulation. Check with your host system for information on how to set these options correctly, as proper settings vary from system to system.



The buttons on the left work for all terminal modes. The Application Mode Keypad button controls whether using the keypad sends numbers or control sequences. *Line wrap* controls whether long lines are wrapped or truncated. On the right, Appl. Mode Cursor Keys controls the codes sent by the cursor keypad. Insert Characters Mode is an advanced mode that lets characters be inserted into a line. NewLine Mode causes a Return to transmit a Return/Linefeed, and moves the cursor to the beginning of a new line when you receive a linefeed. Swap Backspace and Delete reverses the functions of these two keys on the keyboard; it is primarily designed to facilitate connecting to Digital VAX systems.

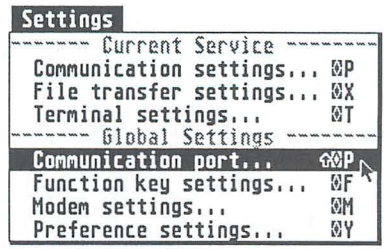
If you have GDOS loaded, the Font To Use For Terminal button will allow you to select a GDOS font. Use the Point Size text field to select a size. The IBM Character Set button controls whether the current font uses the IBM or Atari Character set. See the *Technical Information* section for more information on these features.



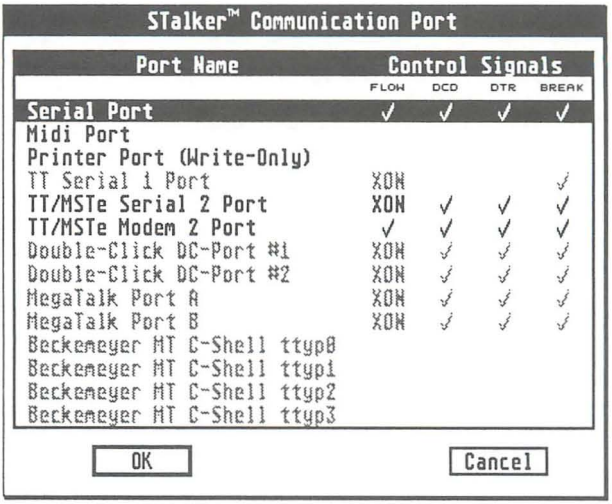
There are 4 ways to exit from this dialog box. The OK button puts the changes you made to the box into effect until you change them again or dial using the Autodialer menu. The Change All button puts the changes into effect and copies them to all entries in the Autodialer. If you have used the Autodialer to call a system, the Change (name) button changes only the Autodialer entry you are using to reflect the new parameters. Finally, the Cancel button exits the dialog box and eliminates all changes.

7.4.4. Communication port... [Shift][Alt][P]

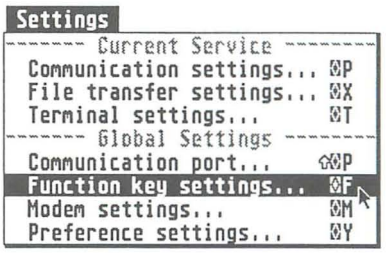
This option lets you choose the port to which your modem is connected. STalker can support any of the ports on the Mega STe and TT/030, plus Double Click Software's *DC-Port*, Gadgets by Small's *MegaTalk*, and Beckemeyer's simulation of multiple ports in software. Left-click on the name of the port you want to use.



If you have an ST/TT with more than one port, and you have multiple modems, it is possible to keep multiple copies of STalker in memory, each one talking to a different port/modem combination. In addition, if you own STeno, you can keep multiple copies of that program in memory and have each copy of STeno "attached" to a different copy of STalker. See the chapter entitled *Keeping Multiple Copies of STalker and STeno in memory* for further information. The *Control Settings* columns tell you which ports support which signals.


STalker™ Communication Port
Port Name
Control Signals
FLOW DCD DTR BREAK
Serial Port ✓ ✓ ✓ ✓
Midi Port
Printer Port (Write-Only)
TT Serial 1 Port XON ✓
TT/MSTe Serial 2 Port XON ✓ ✓ ✓
TT/MSTe Modem 2 Port ✓ ✓ ✓ ✓
Double-Click DC-Port #1 XON ✓ ✓ ✓
Double-Click DC-Port #2 XON ✓ ✓ ✓
MegaTalk Port A XON ✓ ✓ ✓
MegaTalk Port B XON ✓ ✓ ✓
Beckemeyer MT C-Shell ttyp0
Beckemeyer MT C-Shell ttyp1
Beckemeyer MT C-Shell ttyp2
Beckemeyer MT C-Shell ttyp3
OK
Cancel

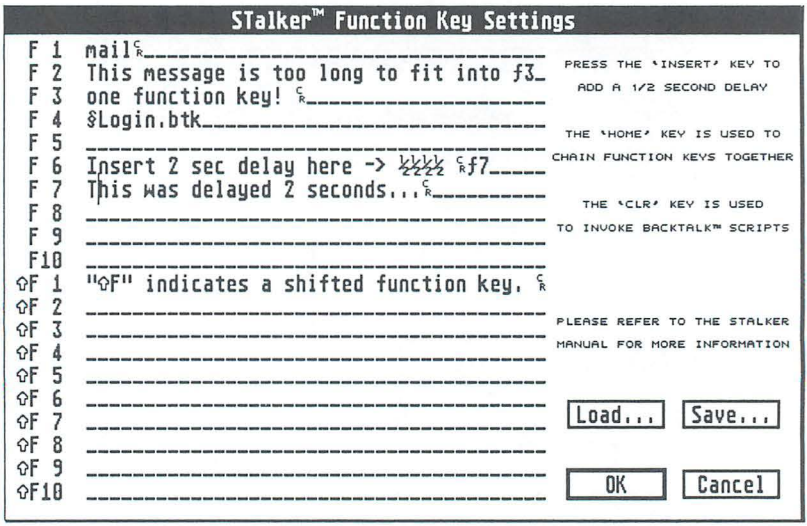
7.4.5. Function key settings... [Alt][F]



This option brings up a dialog box which allows you to program the function keys ([F1] through [F10]) so that when you press them, they transmit a message of your choice to the host computer. There are 20 text fields in the dialog box, allowing individual programming of keys [F1] through [F10] and [Shift][F1] through [Shift][F10]. Once the function keys are programmed, press the appropriate key, or [Shift] and that key, to transmit your message.

To program in a 1/2 second delay at any point in the transmission of the message, use the [Insert] key on the keyboard. The symbol "½" will appear to indicate the pause.

If your desired message does not fit in the text field allowed for each function key, you can chain the keys together. This means that STalker can send another key's message at any point within the first key's message To use chaining, at any point in the text field type the [Clr/Home] key (the symbol "*f*" will appear) followed by an f-key number from 1 to 20. 1 to 10 represent the ten function keys, while 11 through 20 represent the keys plus the [Shift] key. Do not put a space between the "*f*" and the number. When the function key's message is sent, the *f#* code will be substituted by the chained function key's message. You can chain keys within already chained keys ("nesting"). If, for example, you wanted to send function key 1 followed by the number 7 you would need to enter "*f*017", as "*f*17" would be interpreted as function key 17.


STalker™ Function Key Settings
F 1 mail
F 2 This message is too long to fit into f3_
F 3 one function key!
F 4 §Login.btk
F 5
F 6 Insert 2 sec delay here ->
F 7 This was delayed 2 seconds...
F 8
F 9
F10
⇧F 1 "⇧F" indicates a shifted function key.
⇧F 2
⇧F 3
⇧F 4
⇧F 5
⇧F 6
⇧F 7
⇧F 8
⇧F 9
⇧F10
PRESS THE 'INSERT' KEY TO ADD A 1/2 SECOND DELAY
THE 'HOME' KEY IS USED TO CHAIN FUNCTION KEYS TOGETHER
THE 'CLR' KEY IS USED TO INVOKE BACKTALK™ SCRIPTS
PLEASE REFER TO THE STALKER MANUAL FOR MORE INFORMATION
Load...
Save...
OK
Cancel

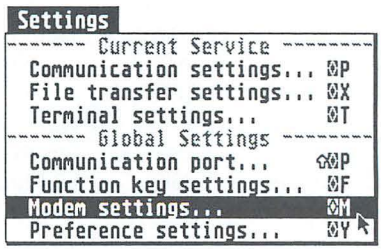
If your message requires a significant portion of the available function keys, you may find it more efficient to write a BackTALK script rather than programming the keys. To cause a function key to invoke a BackTALK script, place the cursor in the field of the key you want to program, then press [Shift][Clr/Home]. A "§" symbol will appear in the field. Then, put the name of the BackTALK script you wish to run. Note that the directory in which BackTALK scripts are located is set using the *Preference Settings...* command on the Settings menu.

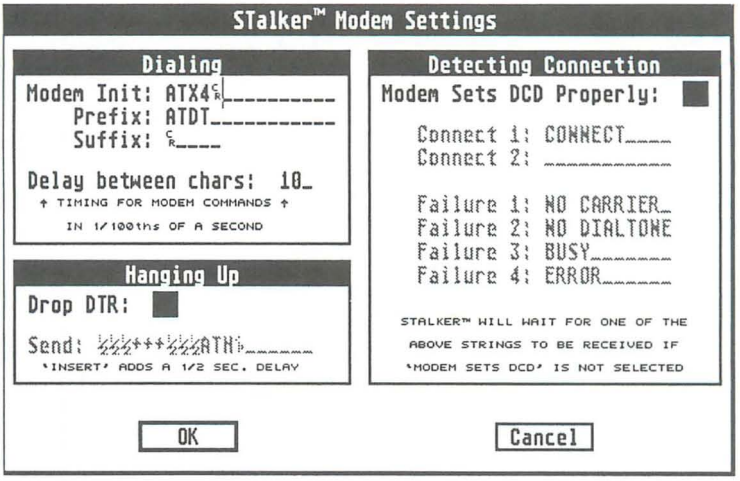
The *Load...* and *Save...* buttons bring up the GEM file selector so you can load and save your function key settings. Do not forget to save after editing. If you exit STalker without saving, you will lose all your work! *Save Configuration...* will cause your keys to be loaded automatically.

The OK button exits the dialog box and implements your changes. The Cancel button exits the box and ignores all changes.

7.4.6. Modem Settings... [Alt][M]

This option displays a dialog box which lets you configure your modem settings. If you are using a Hayes compatible modem, you should not need to change anything in this dialog box. There is only one exception: if you do not have Touch Tone dialing, change the PREFIX text field from "ATDT " (tone) to "ATDP" (pulse).





If you are not using a Hayes compatible modem, consult your modem manual to determine the correct settings are for the entries in this dialog box.

The options in the Dialing section of the dialog box allow you to configure STalker's autodialer to work with your modem. The Modem Init text field is sent to the modem each time the autodialer dials. Next, STalker sends the Prefix field's text, followed by the number programmed into the autodialer, and finally, the text in the Suffix string. Carriage Returns are displayed as a small "c/r", and are entered with [Control][M]. If you don't have enough space, you can use function key chaining in these fields. If you leave both Prefix and Suffix strings empty, STalker will assume you are using a manual-dial modem, and will display an alert box instructing you to dial the telephone.

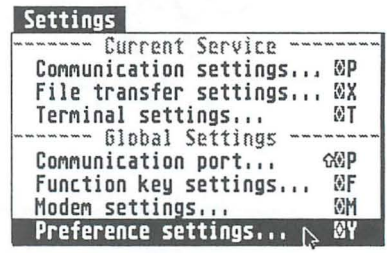
If you find that STalker doesn't seem to be able to dial properly, then it's possible that your modem isn't handling the DCD (Data Carrier Detect) line correctly, as STalker normally monitors this line to determine if a dialing attempt was successful. This can be solved either by changing your modem settings (if you have a Hayes-compatible modem, send it the "AT&C1" command, then save the new configuration) or by left-clicking in the Modem sets DCD properly box to turn it off. When DCD handling is off, STalker will examine the modem's responses for one of the success (connect 1 and 2) or failure text fields. Enter the data into these fields according to your modem's manual.

STalker also uses the RS-232 DTR (Data Terminal Ready) signal in order to hang up the phone line. Make sure that your modem is not ignoring the DTR signal or the Hang-up phone line command will not work. For Hayes compatible modems, this is accomplished with the "AT&D1" command. If your modem is not Hayes compatible, consult your modem manual. If your modem will not hang up when the DTR signal is dropped, left-click on the Drop DTR box to turn it off; this will cause STalker to send the text in the field directly under the button rather than dropping the DTR signal

when you issue a hang-up command. Consult your modem manual to see what to enter in this field. You can use function key chaining as previously described.

The OK button exits the dialog box and implements your changes. The Cancel button exits and ignores any changes. After changing the settings, use *Save Configuration* from the File menu to save your changes.

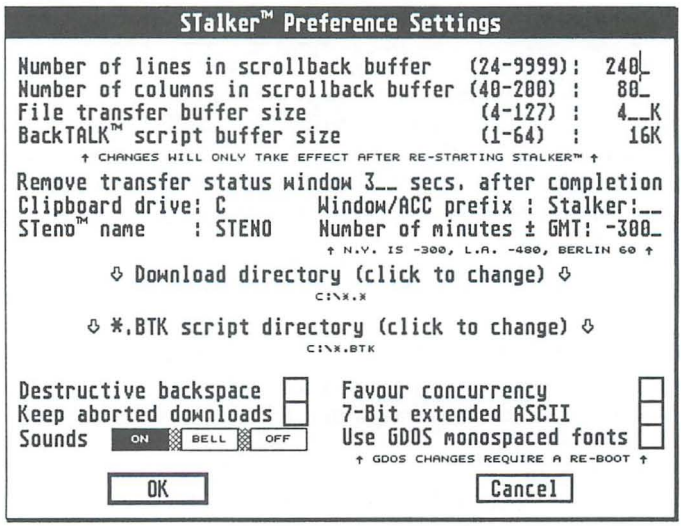
7.4.7. Preference Settings... [Alt][Y]



The Preference Settings dialog box lets you change various settings. After making any changes to this box, save your configuration to make the changes permanent.

The first two text fields, Number of lines in scrollback buffer and Number of columns in scrollback buffer, let you set the size of the virtual screen. The larger the numbers you enter here, the more memory STalker will use.

The File Transfer Buffer Size sets the size of the buffer (memory area) STalker uses when doing uploads and downloads. Increasing the size of the buffer reduces the number of disk accesses STalker will make during a file transfer operation. Buffer size is not overly important for hard disk owners since hard disk I/O is so fast. Floppy users will need to experiment to find a value that balances the frequency of disk access with the time required for each access. Too large a buffer may result in retries during downloads if the time required to save or load the buffer is too long.


STalker™ Preference Settings
Number of lines in scrollback buffer (24-9999): 240
Number of columns in scrollback buffer (40-200) : 80
File transfer buffer size (4-127) : 4K
BackTALK™ script buffer size (1-64) : 16K
↑ CHANGES WILL ONLY TAKE EFFECT AFTER RE-STARTING STALKER™ ↑
Remove transfer status window 3 secs. after completion
Clipboard drive: C
Window/ACC prefix : Stalker:
STeno™ name : STENO
Number of minutes ± GMT: -300
↑ N.Y. IS -300, L.A. -480, BERLIN 60 ↑
Download directory (click to change)
C:*.*
*.BTK script directory (click to change)
C:*.BTK
Destructive backspace
Keep aborted downloads
Sounds
ON
BELL
OFF
Favour concurrency
7-Bit extended ASCII
Use GDOS monospaced fonts
↑ GDOS CHANGES REQUIRE A RE-BOOT ↑
OK
Cancel

BackTALK Script Buffer Size lets you set the maximum size for a script.

Note: After changing any of these four fields, you must save your configuration and restart STalker for the changes to take effect.

The Remove Transfer Status Window text field lets you enter a number which determines how long the status box displayed during a file transfer operation stays on the screen after the transfer completes.

The Clipboard Drive field lets you choose the drive to which STalker writes the temporary data files (SCRAP.TXT) used by the clipboard.

Note: If the drive letter Z is entered in the field, STalker will use the new Atari standard method for accessing the clipboard. Note that some older programs (including STeno 2.00) are not aware of this method, and may experience difficulty when trying to Paste text which was written by STalker when drive Z is used.

The Window/ACC Prefix field lets you customize the name which is displayed in the Desk Accessories list and at the top of the window for this copy of STalker. Since it is possible to have more than one copy of STalker in memory at a time, each one connected to a different port/modem combination, this option lets you determine which copy is which by giving each copy of STalker a unique name.

The STeno Name field lets you enter the program name under which STeno is running. Normally, this will be STENO. However, if you are running multiple copies of STalker, you may also wish to have multiple copies of STeno as well. (See Keeping Multiple Copies of STalker and STeno in memory for details.) Each copy of STeno will be run using a unique filename (such as STENO1.ACC, STENO2.ACC, etc), and you should enter into the STeno Name field the filename of the particular copy of STeno you want to associate with this particular copy of STalker.

Enter into the Number of Minutes +- GMT field the number of minutes that your timezone is offset from GMT (Greenwich Mean Time, also called Universal Coordinated Time). In the United States, Eastern Standard Time is -300, Central Time is -360, Mountain Time is -420, and Pacific Time is -480. Berlin, Germany, is +60. This number is used to correct timestamps in YModem and ZModem transfers. See the *Technical Information* chapter for further details on timestamps.

The Download Directory lets you set the default path for downloads (that is, the path initially displayed in the GEM file selector when you issue a download command). This path is also used when a ZModem transfer initiates automatically. To change this setting, left-click anywhere on the words Download Directory. The GEM file selector will appear. Choose the path you want, then exit with the OK button.

The *.BTK Script Directory function lets you set the default path for BackTALK scripts (that is, the place in which STalker looks for BackTALK scripts). To change this

setting, left-click anywhere on the words *.BTK Script Directory. The GEM file selector will appear. Choose the path you want, then exit with the OK button.

If you elect to Keep Aborted Downloads, any data received before a download is aborted will be kept. Normally, this option is off and the data is only kept upon successful completion of the download. In order to use the GDOS Monospaced Fonts option, you must have GDOS or a compatible program loaded. Fonts are supplied on the STalker 3 disk. If you are running STalker as a desk accessory, you mus turn this option on, save the configuration, and then reboot the computer before STalker will allow GDOS fonts to be used.

If 7-bit Extended ASCII is selected, a SHIFT-OUT (Control N) character tells STalker to interpret subsequent characters as extended ASCII codes until a SHIFT-IN (control O) is received. If this option is not selected, SHIFT-IN and SHIFT-OUT are ignored. If you receive a spurious SHIFT-IN with this option on and start seeing strange characters, you can return to normal by pressing [Alt][Esc]. This option only affects VT100 and PC-ANSI emulation modes.

The Sounds buttons control the sound effects used by STalker. On plays the full range of tunes. Bell gives you a simple tone. Off causes STalker to briefly invert the menu bar instead of making a sound. This setting affects the "Play_Sound()" BackTALK routine similarly.

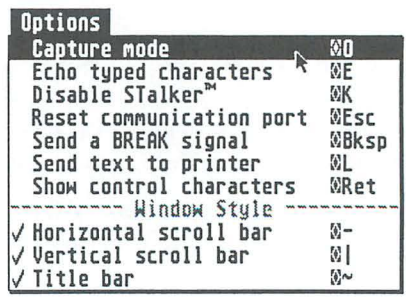
The Favour Concurrency option causes STalker to look for GEM events more often than normal during binary file transfers. This slows down file transfer speed slightly, while improving the system's response time to GEM events. When this option is off and a baud rate of 9600 or greater is being used, very few GEM events will occur. This is so that the maximum throughput can be achieved.

7.5. Options Drop-Down Menu

All the entries on the **Options** menu may be toggled on and off by selecting them. When an option is turned on, a check-mark will appear in the menu to the right of that option.

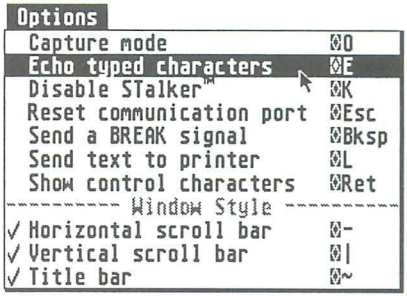
7.5.1. Capture Mode [Alt][O]

This option is only available if STeno is loaded. When capture mode is on, incoming text is sent to STeno, which appends it to the file being edited. This mode is indicated by a black dot which appears to the left of the window title text. See the *Technical Information* chapter for further information.



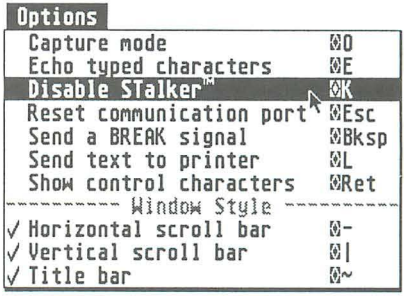
7.5.2. Echo Typed Characters [Alt][E]

This option toggles between Full and Half Duplex. When this option is turned on, you are in Half Duplex mode, when off, Full Duplex. This option does the same thing as opening the Communication Settings dialog box and changing the Full/Half Duplex buttons. If you change the settings using the dialog box, the *Echo Typed Characters* menu option will change accordingly, from checked to unchecked or vice versa. Similarly, if you change the settings using the menu, the dialog box settings will change.



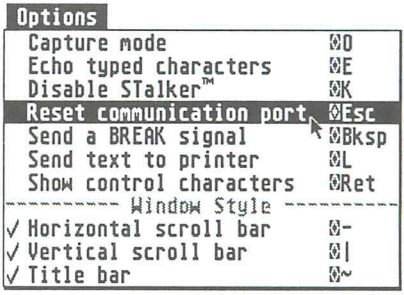
7.5.3. Disable STalker [Alt][K]

This option disables STalker, preventing it from sending or receiving data. To turn STalker back on, choose this option again. When STalker is disabled, you will not be able to execute any commands having to do with sending or receiving data. This option is useful if you want to run another GEM based communications program while STalker is loaded, as it stops STalker from reading data from the communications port.



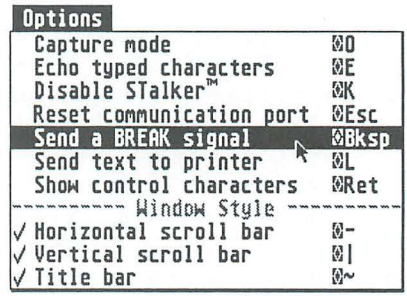
7.5.4. Reset Communication Port [Alt][Esc]

This command resets the current communications port. This is helpful if the port locks up, something that happens more frequently on TOS 1.0 and 1.2 based machines. See the *Technical Information* chapter for further details.



7.5.5. Send a BREAK Signal [Alt][Backspace]

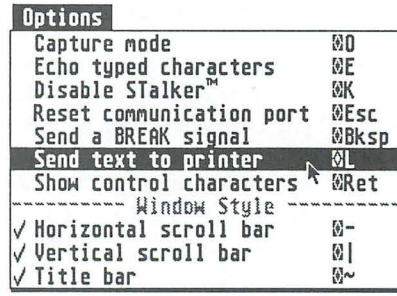
This command transmits a BREAK signal. Some systems use this special signal to stop certain operations. You can also use [Control][Undo] to send a BREAK signal.



7.5.6. Send Text to Printer [Alt][L]

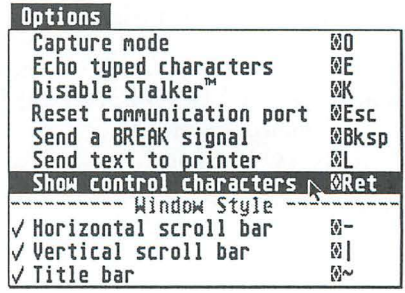
When on, this option echoes any text to your printer as it is received. Control and Escape sequences are filtered out and each line is printed after the cursor moves off it.

This mode is indicated by a dot which appears to the right of the window title text.

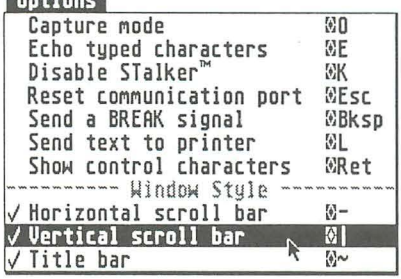


7.5.7. Show Control Characters [Alt][Return]

Terminal emulation uses invisible control characters to select colors, move the cursor, etc. By turning this option on, you can see these characters. Their appearance will vary depending on the currently selected font.



7.5.8. Window style

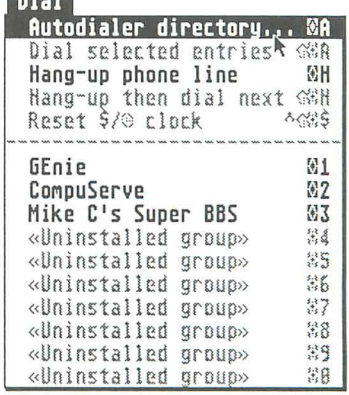


- Horizontal Scroll Bar [Alt][-] (dash key left of [=])
- Vertical Scroll Bar [Alt][|] (key under [Delete])
- Title Bar [Alt][~] (key next to [Backspace])

These three options turn the corresponding window border element on or off. A check mark displayed to the left of the menu option indicates that the element is being displayed. Remember that turning off either or both scroll bars will also turn off the resize gadget, while turning off the title bar will also turn off the close and full size gadgets. See the *Using STalker* chapter for a discussion of window gadgets and ways to manipulate the STalker window even when some gadgets are hidden.

7.6. Dial Drop-Down Menu

7.6.1. Autodialer Directory [Alt][A]

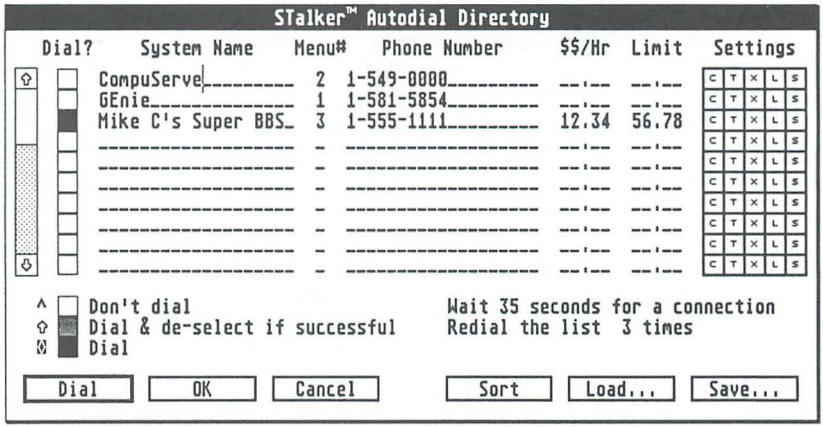


This entry brings up the Autodialer Directory dialog box, which enables you to program up to 30 entries to be automatically dialed. Each entry contains its own individual data for system name, phone number, communication settings, file transfer settings, etc.

As only 10 entries at a time are displayed, use the vertical scroll bar on the left side of the dialog box to scroll up and down the list of entries.

The Dial? button on the far left of each Autodialer entry works with the Dial Selected Entries command from the Dial menu or the Dial exit button at the bottom of the box. A single left-click puts this entry into the list of entries to be dialed. This choice, displayed by making the button light gray (or green for color systems), means that the entry will be removed from the dialing list after a successful connection. A second left-click (not a

double click), shown by a black button, also puts this entry into the list, and it stays in the list even after a successful connection. A third left-click removes the entry from the list. Finally, a double left-click on the button causes this entry to be dialed immediately. To interrupt dialing, press the [Undo] key.



In the System Name field, enter the name of the computer or service this dialer entry calls.

The Menu# field holds a number from 0 through 9, corresponding to the 10 entries at the bottom of the Dial drop-down menu. When you enter a number into this field, the system name will appear in the corresponding entry in the Dial menu, and you can dial the entry by selecting it off the menu. If you give multiple entries the same number, each entry will be dialed in sequence until one answers.

Enter the complete phone number for the dialer entry into the Phone Number field, including 1-Area Code if necessary.

The $$/Hr field can hold the cost per hour of the dialer entry. If you enter a number into this field, when you are connected STalker will display a "cash clock" showing how much you've spent since you logged on. If the service is free, leave this field blank and STalker will show you a connect time clock instead.

The Limit field holds a dollar value. If you enter values into the $$/Hr and Limit fields, STalker will give you an audible warning when you exceed your dollar limit.

The five buttons at the right of each dialer entry control the various settings for that entry. In order, these buttons are:

C Communications Settings
T Terminal Emulation Settings
X File Transfer Settings
L Auto Logon Sequence
S Auto-Execute BackTALK Script

Each entry in the Autodialer has its own independent settings for each option. When you click on one of the buttons, the corresponding dialog box will appear, allowing you to change the settings for that entry. The C, T, and X options bring up dialog boxes identical to the ones from the **Settings** menu, except that they only have OK and Cancel exit buttons. Refer to the instructions for the **Settings** menu for detailed instructions on these dialog boxes. Remember that changing the settings in these boxes does not change STalker's current settings; the changes you make are only associated with the corresponding Autodialer entry, and are implemented only when you autodial that entry.

The L button brings up a dialog box which enables you to enter an Auto-Logon sequence. This allows you not only to dial automatically, but to log into the host system without even touching the keyboard. Examine this box, and you will see eight Host Prompt fields, each followed by a "text to respond" with field. STalker uses these fields as follows: After a connection is established, STalker waits for the data in the first Host Prompt field to be received. When it is, the corresponding Text to respond with data is transmitted. STalker then waits for the second host prompt field, and so on. If the Host Prompt field is left blank, STalker will immediately respond with the correct string. You can insert pauses in each response field and at the beginning of the prompt field by pressing the [Insert] key once for each 1/2 second delay. Use [Control][M] to insert carriage returns at the end of each response. Don't forget the Returns!

You only need the "tail end" of each prompt in the Host Prompt field. Don't worry if it all doesn't fit. For instance, if the host sends "Password/ID?", all you need is enough to differentiate this message form others, such as "ID?". A single "?" may even be enough in some cases.

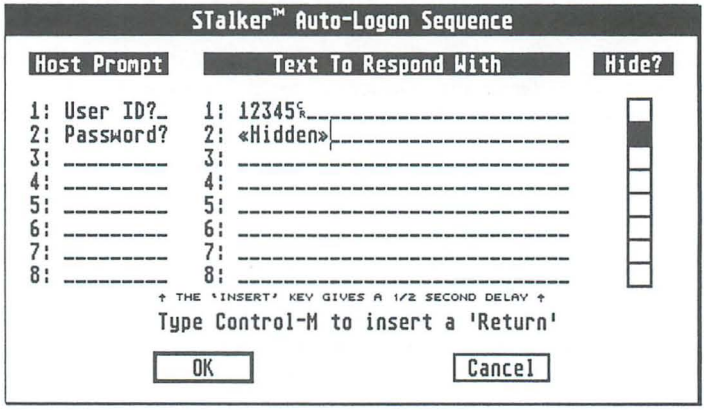
The Hide button causes a response field to be hidden upon subsequent views of this dialog box. In addition, a hidden field will not be echoed to the screen, even in half-duplex mode. This will prevent others who may have access to your computer from looking at your passwords. Once a field is hidden, you will never be able to view its contents.

As an example of an Auto-logon sequence, suppose your host computer displayed "USER ID?" upon connection, and at this point, you enter "12345" and press return. Suppose the computer would then respond with

USER ID OK

ENTER PASSWORD? and you enter here "XYZZY" and press return.

Your Auto-logon sequence would look like this:



When you redisplay this box, XYZZY will be hidden.

The S box lets you specify a BackTALK script to execute upon login. The script is located in the directory specified in the *Preference Settings...* dialog box.

At the bottom of the Autodialer Directory box are two text fields. The Wait XX seconds for connection, where "XX" is a number in seconds, tells STalker how long to wait for a connection before giving up. The Redial the list XX times, where "XX" is a number from 0-99, tells STalker how many times to dial the list of selected entries when you choose the Dial exit button or *Dial Selected Entries...* from the Dial menu. If you enter 0 into this field, STalker will continue to dial every entry on the list until one answers.

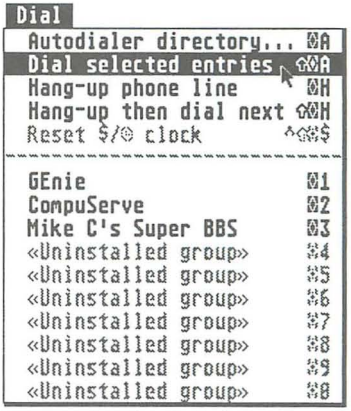
In the lower right-hand corner of the Autodialer Directory box are three buttons which enable you to sort, load, and save your work. The Load... and Save... buttons display the GEM file selector so you can enter a filename for your data. While the default Autodialer entries are saved in the STALKER.INF file, this allows you to have unlimited additional Autodialer files. Use Save Configuration to make the current autodialer settings load automatically.

Finally, the exit buttons are at the bottom left of the box. The Dial button exits the box and begins dialing selected entries. The OK button exits the box and keeps any changes that you have made (though if you want to save them when you shut off the computer, select [Alternate][G] to save the default configuration file). The Cancel button exits the dialog box and ignores any changes made.

7.6.2. Dial Selected Entries... [Shift][Alt][A]

When you display the Autodialer Directory dialog box, you can select entries to be dialed by left-clicking on the box to the immediate left of each dialer entry.

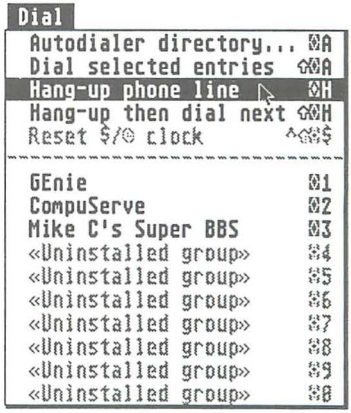
When you choose this menu option, selected entries will be dialed, one at a time, just as if you had displayed the Autodialer Directory dialog box and exited with the Dial button.



7.6.3. Hang Up Phone Line... [Alt][H]

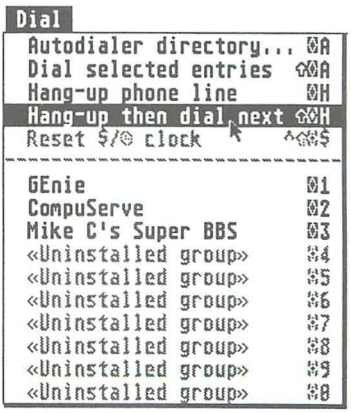
This option hangs up the phone line, using the method specified in the Modem Settings dialog box. This option also causes the communications/transfer/terminal settings to revert to their default values.

The default values are the settings that were in effect at boot time, or at the time of the most recent *Save Configuration* command, if one has been issued since boot time.



7.6.4. Hang-up then Dial Next [Shift][Alt][H]

This option hangs up, then continues to dial selected entries in the list, starting with the next one in the sequence, until one answers. The effect is the same as selecting [Alternate][H] followed by a [Shift][Atlernate][A].



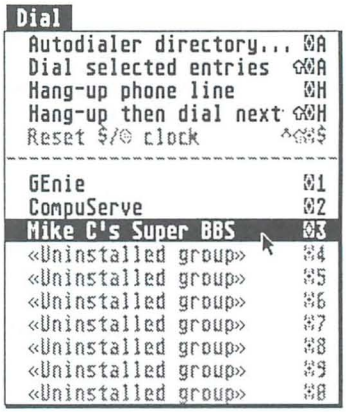
7.6.5. Reset $/Time Clock [Control][Shift][Alt][$]

This option will reset the connect time/dollar clock to zero.

7.6.6. Dialing Groups [Alt][0] through [Alt][9]

Note: Use numbers on main keyboard.

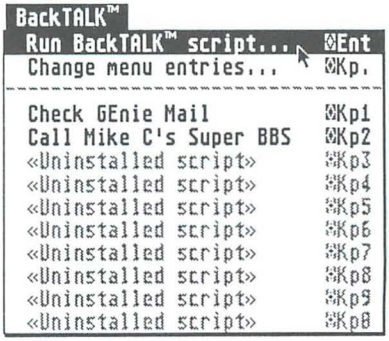
If you entered a number from 0-9 into the Menu# field of the Autodial Directory box, that entry will be displayed in the corresponding field (0-9) of the Dial drop-down menu, and you can dial that entry by choosing it from the menu. If you had more than one entry with the same number, the text displayed in the menu will be the name of the first dialer entry with that number, and the entries will be dialed in order until one answers.



7.7. BackTALK Drop-Down Menu

7.7.1. Run BackTALK Script [Alt][Enter]

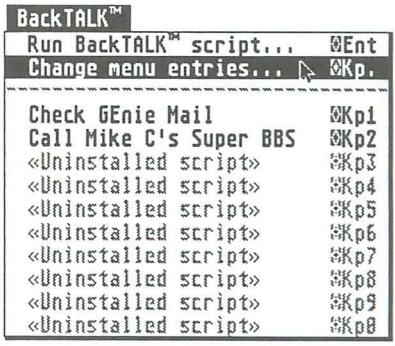
When you choose this option, you will see the GEM file selector. Choose a BackTALK script to execute. The default directory shown will be the one specified in the Preference Settings dialog box.

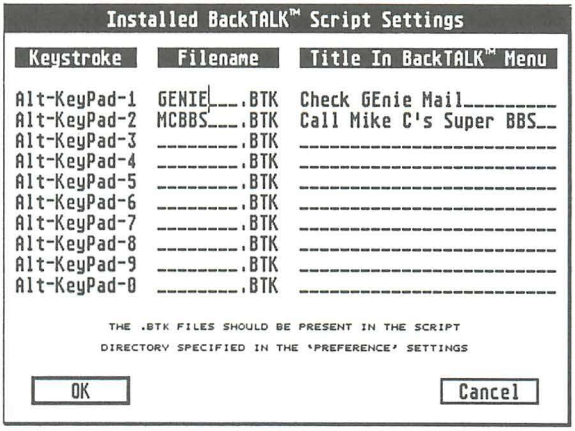


7.7.2. Change Menu Entries... [Alt][.]

Note: Use the period on the numeric keypad.

This option displays a dialog box which enables you to enter up to 10 BackTALK scripts to be displayed in the BackTALK drop-down menu. For each entry, enter the filename and the name you would like displayed in the menu. BackTALK files must be located in the directory specified in the Preference Settings dialog box. Clicking on the Filename field will display the GEM file selector.

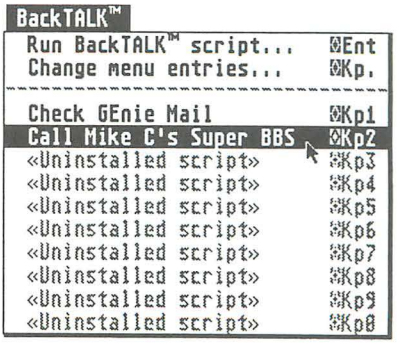




7.7.3. Scripts [Alt][0] through [Alt][9]

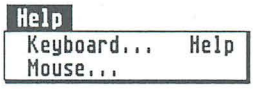
Note: Numbers on the keypad only. Can also use [Control] instead of [Alt].

After you have specified entries using the Change Menu Entries dialog box, you can execute a BackTALK script by selecting it from this section of the menu. Uninstalled entries will be grayed out and cannot be selected.



7.8. Help Menu

The two entries in the **Help** menu display dialog boxes with information on basic keyboard and mouse commands. Left-click or press any key to exit the Help boxes.



8. Using STeno with STalker

8.1. Keeping Multiple Copies of STalker and STeno in Memory

Some Atari computers, such as the TT, have multiple serial ports. Even if you only have one serial port, you may own a product such as Double-Click Software's *DC-Port*, which gives your computer extra ports. If you have these extra ports, you can connect more than one modem to your computer and connect to more than one host system at the same time. STalker is only designed to talk to one port at a time. Fortunately, as STalker is also designed to run at the same time as other programs, there is a simple solution. If you have more than one port, you can keep more than one copy of STalker in memory, each one connected to a different port!

To do this, simply make copies of the STalker desk accessory file (STALKER.ACC) keeping the copies in the same directory. You can use any name you want as long as you keep the ".ACC" extension, but it is best to use STALKER1.ACC, STALKER2.ACC, etc. since each copy will then use STALKER1.INF, STALKER2.INF, etc., and will label itself with a name and number, so you can tell which accessory is which. If you use other names, they all will load STALKER.INF, and you will have to manually rename each accessory as explained in the next paragraph. When you reboot (reset) your computer, you will see the STalker entry on the desk menu as many times as you made copies of the file. (Remember, without special software there is a limit of 6 desk accessories.)

Open each copy of STalker and use the *Communication Port* dialog box to set each copy to a different port. Then, open the *Preference Settings* dialog box and use the Window/ACC Prefix text field to give each copy of STalker a unique name. This name will be displayed in the Desk (or Atari "Fuji" Logo) menu and at the top of the window. (This is not necessary if you numbered your accessory's filenames, as the windows will automatically be numbered.) Finally, use *Save Configuration* to save each copy's setup in a unique file. If you do not use *Save Configuration As...*, but you did number your accessory files, the name of the setup file will correspond to the name of the ".ACC" file; STALKER.ACC will save to STALKER.INF, STALKER1.ACC will save to STALKER1.INF, and so on).

As STalker can be run as a stand-alone program, if you have no other desk accessories, you can have as many as seven copies of STalker running at once - six accessories plus the stand-alone program copy.

STalker also communicates with STeno, Gribnif Software's desk accessory text editor. If you are running multiple copies of STalker, you may also want independent copies of STeno. Install the copies of STeno as you did for STalker, keeping in mind the six accessory limit. Then, when you set up each copy of STalker, change the text field in the *Preference Settings* dialog box where it says STeno Name to the filename of one of the copies of STeno. Make sure that each copy of STalker communicates with an individual copy of STeno. Finally, activate each copy of STeno and choose *Preferences...* from the **Options** menu. Enter the filename of the copy of STalker you want this copy of STeno to communicate with into the field labeled STalker to communicate with and change the field labeled Window/ACC Title Prefix to a unique name (this field is STeno's equivalent of STalker's Window/ACC Prefix field in the *Preference Settings* dialog box).

8.1.1. Example

If all this seems confusing, here is an example. Suppose you have an Atari TT030 with two modems, each attached to a serial port. You want to have two copies of the desk accessory version of STalker and two copies of the desk accessory version of STeno in memory. The first STalker should talk to modem #1, while the second talks to modem #2. Each copy of STalker should have an independent copy of STeno communicating with it.

First, open a window to the directory where your desk accessory files are located. Install a copy of STalker and a copy of STeno the usual way. Next, as you can only have a maximum of six accessories, you may need to make room for two additional accessories by changing the extension of existing accessories from ".ACC" to ".ACX".

Next, drag the STALKER.ACC file within the directory window and drop it (not into a folder, however). You will see a "Name Conflict During Copy" message. Change the STALKER.ACC text field to STALKER2.ACC and exit the box with the COPY button. Do the same thing with the STENO.ACC file, and change the name to STENO2.ACC. After you are done, reset your computer.

If you look under the **Desk** menu, you will notice that there are now two copies each of STalker and STeno. Open up the topmost copy of STalker, open the *Communication Port* dialog box, and choose the first modem port. Exit the dialog box and close this copy of STalker.

Next, open up the copy of STalker which is the bottommost on the Desk menu. Open up the *Communication Port* dialog box and choose the second modem port. Exit the dialog box. Then, open the *Preference Settings* dialog box. Change the text field

labeled Window/ACC Prefix to "STalker 2:" (do not type the quotes). Change the field labeled STeno Name to "STENO2". Exit the dialog box.

Now, open the copy of STeno bottommost on the Desk menu. Choose *Preferences...* from the Options menu. Enter "STALKER2" into the field labeled STalker to communicate with and change the field Window/ACC Title Prefix to "STeno 2:".

Unfortunately, the order in which the various programs show up on the desk menu is not necessarily the order in which they are stored on the disk. Therefore, you cannot assume that STalker and STeno are communicating with each other and STalker2 and STeno2 are communicating. It is also possible that you may have STalker talking to STeno2, and STalker2 talking to STeno. Therefore, the final step is to open one of the two copies of STeno, and choose *Arrange STalker & STeno Windows* from the STalker menu. See which copy of STalker and which copy of STeno arrange themselves on the screen. This shows you which copy of STalker is connected to which copy of STeno. If you have it set up as STalker-STeno2/STalker2-STeno, you may wish to adjust the Window/ACC Prefix entries in STalker, and the Window/ACC Title Prefix field in STeno so that the copys' titles match properly.

8.2. Using STeno as a Type-ahead Buffer for STalker

A type-ahead buffer is an area of memory into which you can enter and edit text before you send it to your modem. STeno is designed with special features that make it perfect for use as a type-ahead buffer for STalker.

To use STeno as a type-ahead buffer, open the STeno window and select *Type-Ahead Mode* from the STalker drop-down menu in STeno. You will see a special symbol composed of three horizontal lines displayed at the left margin of the STeno window. This symbol indicates that type-ahead mode is turned on.

While in this mode, as soon as you press [Return], the current line of text in the STeno window will be sent to STalker, which will transmit it over the modem. Before you press [Return], you can edit the line using the left and right arrow keys, the [Delete] key, and the [Backspace] key. This enables you to make sure your line of text is entered correctly before you actually transmit anything. If you type more than one line, STeno will word-wrap normally. When you press [Return], everything you have entered will be transmitted by STalker. The STeno menu entry Send a <CR> after each Line controls whether or not word-wrapped lines are sent with Carriage Returns after them. If the option is on, CRs are sent. If not, STeno sends a single space between each line, making the input look like one big line (great for online conferences!).

When you turn type-ahead mode off by selecting Type-Ahead Mode a second time, the text you sent to STalker is not erased. It remains in the STeno window and it may be edited or saved just like any other text in the window.

If you wish to send pre-existing text to STalker, or type without using type-ahead mode and then transmit the text, enter your text into STeno normally or load a file from disk. Then, define a block of the text you want to send to STalker (see your STeno manual for details). Finally, choose Transmit Line/Selection from the STalker menu in STeno. The block will be sent to STalker, which will transmit it over the modem.

9. Using STalker 3 with NeoDesk

When running STalker as a desk accessory, you can install it as a *NeoDesk Accessory*. This gives STalker some extra capabilities when used with Gribnif Software's Neodesk 3 and NeoDesk 4 desktops.

To install STalker 3 as a *NeoDesk Accessory*, enter the STalker filename in the *Accessories...* dialog box in the *Set Preferences...* menu entry under the NeoDesk **Options** drop-down menu.

Once you have installed STalker 3 as a NeoDesk Accessory, you can place the STALKER.ACC icon directly on the NeoDesk desktop and select *Save Configuration...* to save it permanently.

From this point on, you can do the following from the NeoDesk desktop:

1. You can open the STalker 3 accessory by double-clicking on the STALKER.ACC desktop icon.
2. By assigning a unique desktop icon letter to the STALKER.ACC icon, you can open the STalker accessory directly from the keyboard.
3. You can have STalker automatically execute a BackTALK script by dragging the script icon to the STalker icon and dropping it.
4. NeoDesk windows will be updated automatically when STalker creates a new file during download or when saving a block of text to disk.

10. Technical Information

10.1. Shutdown

Shutdown is a special feature activated by doing a [Control][Shift][Alternate] key combination together with a left-click on the window close box. Doing this, then confirming it at the alert box, will cause STalker to free all of its allocated memory and other system resources, then shut down. It is intended for use just before a change of resolutions, to prevent loss of system RAM. This is more common in older TOS systems.

10.2. The STalker Clipboard

The STalker clipboard is actually a hidden text file called SCRAP.TXT which is saved in the clipboard directory of the drive specified in the *Preferences...* menu entry. This file is created whenever you use any of the clipboard options. Since STeno uses the same filename for its clipboard, any STalker clipboard usage overwrites the STeno clipboard, and vice-versa. Thus, you can transfer data through the clipboard between STalker, STeno, and any other program using the same clipboard system.

STalker can use two different methods to write to the clipboard. In the "old" method, STalker writes a hidden SCRAP.TXT file to the drive specified in the *Preferences* dialog box. The "new" method is selected by using a *Clipboard Drive* of "Z"; the path is not set directly by STalker, but it is inquired from the operation system. This allows all programs which are aware of the new standard to use the same location for clipboard files.

10.3. Y/Z Modem Information

In YModem and ZModem, the sender transmits the file name, and optionally the file size, modification date/time, the file "mode" (permissions etc. a la UNIX) and a serial number. Most YModem implementations send the file size, some send more; it is up to the receiving program to decide what to do with the data. STalker sends the size, date, and mode. The mode will be 0 unless the file is marked as 'read only', in which case the mode will be octal 444 (Unix mode for a read-only file).

As per the Y/ZModem specification, filenames are converted to lowercase before transmitting, and the slash character (/) is used to delimit directories. If the file being sent happens to contain a "/" in its name, STalker changes it to an underscore character.

10.4. Timestamps in Y/ZModem

The modification time is sent as the number of seconds since January 1, 1970 Universal Coordinated Time (GMT). Since the Atari operating system has no concept of time zone, STalker allows the user to enter the system's "offset" from GMT in minutes. This amount affects the conversion of time values between GMT and Atari format. A value of zero means that the value will be converted but not altered. Non-zero values are subtracted in the Atari-to-GMT conversion, and added in the GMT to Atari conversion. If you transfer files with a system that handles time zones correctly (like a Unix host), it is a good idea to set your GMT offset. If you transfer files with systems that do not handle time zones (such as most BBS systems), then you can leave it set to 0. There is no sure way to tell what any other system is doing other than to look at the timestamps you get on received files and see if they are what you expected.

10.5. File Transfer Efficiency

The upload and download status boxes display an "efficiency" rating showing the efficiency of your transfer speed, where 100% is the theoretical maximum speed (baud rate divided by 10). This efficiency is dependent on the protocol being used as well as the speed of the sending and receiving computers. The protocol affects efficiency since extra characters may be transmitted to perform error checking, etc. Protocols like XModem that require the sender to stop and wait for the reciever to acknowledge each block reduce the efficiency since there are small periods of "dead time" where nothing is happening. STalker calculates efficiency by looking at how many bytes of actual data have been transmitted in the elapsed time, not including additional bytes for error checking, etc.

While ZModem is one of the most efficient protocols and you should try to use it as often as possible, there are cases where it can become bogged down (mostly due to its complex error correcting protocols). This usually happens if you are dealing with a very noisy phone line. Under this situation, you might want to try YModem, which is able to more efficiently cope with very noisy phone lines.

10.6. Capture Mode

Capture Mode lets you send incoming text directly to STeno. If you do not have STeno, the Receive Text File command is a good alternative. There are differences between the two methods, however, as follows:

1. *Capture Mode* requires STeno.
2. *Capture Mode* filters out control characters and escape sequences by sending lines of text only when the cursor moves to a new line via a line feed or formfeed. If the

cursor moves off the line due to a cursor positioning command, the line will not be sent to STeno.

3. *Receive Text File* copies every character received from the host computer, except for ASCII Null, to the disk file. Control and escape sequences are not filtered out. This makes it an effective way to capture a session along with the control sequences to be replayed later or used to test other terminal emulators.

4. *Receive Text File* uses the file transfer buffer, so you cannot perform another file transfer while it is active. You can, however, perform a file transfer while *Capture Mode* is active.

5. *Receive Text File* is more efficient than *Capture Mode*, since it does not require communiction with STeno, which requires a good amount of system overhead.

10.7. Reset Communication Port

The *Reset Communication Port* command in the **Options** menu has the following effects:

1. It resets the XON/XOFF flag so that the Atari can transmit.

2. The video attributes (bold, underline, etc) are reset. Color is not changed, however.

3. Any escape sequence currently being processed is aborted.

4. The VT100/PC-ANSI character mappings are reset to their default values.

5. The VT100/PC-ANSI scrolling region is reset to the entire virtual screen.

6. Any characters waiting in the port buffer are removed.

10.8. Autodialing Routine

The following is a detailed explanation of the way in which STalker's Autodialer dials.

1. STalker finds the first selected entry in the list and switches the communication port settings to the ones for that entry.

2. STalker sends the modem init string, and then pauses for half a second to let the modem "digest" it.

Then, for each selected entry, STalker will:

1. Switch to the communication settings for that entry.

2. Send the prefix, then the number, then the suffix string to the modem. If the prefix and suffix strings are empty, you will be instructed to dial the phone manually.

3. Pause for 1/4 second to let the modem "digest" the command.

4. STalker will then wait up to NN seconds for a connection, where NN is the number in the *Autodialer* dialog and 0 means to wait forever. The following algorithm is used. Note that the timing begins after STalker sends the command to the modem. It is at this point that the modem begins to dial.

5. If DCD is enabled and carrier becomes active, dialing was successful. Go to step 10.

6. If DCD is enabled and the modem sends some text, STalker assumes it to be a status message such as "BUSY". STalker will read the response from the modem and display it in the terminal window. There will then be a 1/2 second pause to give the modem time to set DCD, then STalker will examine DCD. If it is active, dialing was successful; go to step 10. Otherwise, dialing was not successful; go to step 15.

7. If DCD is not enabled, STalker waits for one of the 6 connect/failure strings to be received. If a connect string is received, dialing was successful; go to step 10. If a failure string was received, dialing was not successful; go to step 15.

8. If dialing was successful, the special tune is played and the window title is updated accordingly.

9. The file transfer and terminal settings for the entry are activated.

10. The *Logon Prompt/Response* strings, if any, are used.

11. The *Cash/Time Clock* is reset and turned on.

12. The *BackTALK Script To Be Executed*, if any, is executed.

13. If dialing was not successful, STalker sends a Carriage Return to the modem and then pauses for 1/4 second. This will cause Hayes-compatible modems to abort the current command.

14. If DCD is enabled and carrier was not present when dialing started, STalker will drop the DTR signal (assuming DTR is enabled and available on the current port). This will get most modems to abort dialing and return to command state, and is necessary for non Hayes-compatible modems that are not affected by step 15.

15. STalker will then pause for 3/4 second and then return to step 3 for the next selected entry.

Note: If you are dialing a long distance number, particularly with pulse dialing, it may take a long time for the dialing to occur, so you may need to increase the timeout value (wait NN seconds for connection in the *Autodialer* dialog box). If STalker

aborts the dialing attempt before the modem can respond, you can usually assume that the timeout value was too small.

10.9. ASCII Null Character

Typing [Control][2] will transmit an ASCII Null character.

10.10. Using MultiDesk

While STalker will work inside the *MultiDesk* program from CodeHead software, we recommend that if you plan to use STalker with STeno or as a NeoDesk Accessory you keep STalker outside of MultiDesk. This is because when STalker is used in this way it relies heavily on the GEM "pipeline communications" which do not always work reliably from inside MultiDesk. Keeping STalker outside MultiDesk will also help prevent problems if the STalker accessory is directly invoked by the *Invoke STalker* command inside STeno or by the NeoDesk desktop. This is not due to a bug in Multidesk, but to a limitation in the GEM environment which only allows one MultiDesk accessory use the GEM "pipeline" at a time.

10.11. Using the FastLoad Option

You should not use the TOS 1.4 or higher *FastLoad* option with STalker when it is run as an accessory. Using this option can cause unexpected behavior, including system crashes. It is safe to use the *FastLoad* option when running STalker as a program.

10.12. Using PinHead

If you use the PinHead shareware program from CodeHead Software, you MUST make sure that you have a PINHEAD.DAT file in the root directory of your boot disk. Inside this file you should have entered STALKER.ACC (or whatever other file name you are using). This is because when STalker runs as a desk accessory it requires that its memory be cleared before execution. For more information, please consult the PinHead documentation. Not entering the STalker name in the PINHEAD.DAT file is likely to cause unexpected system crashes.

10.13. Using GDOS

In the GDOS folder on your STalker disk, you will find GDOS.PRG and several PC style GDOS fonts. IBMO6.FNT is a 6 point, 8x8 pixel wide font. IBM11.FNT and IBM12.FNT are 11 and 12 point versions that have 8 x 14 and 8 x 16 pixel dimesions, respectively. In addition, there is a PC-ANSI 9 point font, which lets you have 80 columns displayed in a regular screen size.

In addition to the supplied fonts, you can use others. STalker can recognize any monospaced (non-proportionsal) GDOS font in which all regular 256 characters are present. The Typewriter font is an example. A font missing some of the control characters will be ignored.

However, you should keep in mind that GDOS fonts can slow down the text display in older TOS systems.

To use GDOS fonts, you must have GDOS.PRG, FONTGDOS.PRG, NVDI.PRG, or a compatible program in the AUTO folder of you boot disk. Then, you must add the fonts to your ASSIGN.SYS file, as follows:

10.13.1. Sample ASSIGN.SYS File

```
path = c:\GDOS  (or whatever directory your fonts are in)
01p screen.sys ; DEFAULT
    ...other fonts you may have
02p screen.sys ; low res
    ...any low res screen fonts here
03p screen.sys ; medium res
IBM06.FNT
IBM11.FNT
IBM12.FNT
PC-ANSI.FNT
    ...other fonts you may have
04p screen.sys ; high res
IBM06.FNT
IBM11.FNT
IBM12.FNT
PC-ANSI.FNT
    ...other fonts you may have
    ..the rest of your ASSIGN.SYS file
```

11. Index

A

Abort, 41
Action window, 21
ANSI-BBS, 26
Append to clipboard, 22
ASSIGN.SYS, 75
Auto execute Script, 58
Auto logon, 58
Autodialer directory, 42, 57

B

B-Plus, 31
BackTALK, 62
Baud rate, 24
BBS, 27
Binary downloads, 30
Binary file protocol settings, 44
Binary files, 29
Binary uploads, 30
Block action window, 21
Block functions, 20
BPLUS.TTP, 31
BREAK signal, 56

C

Capture Mode, 54, 71
Change of Address, 7
CLI, 38
Clipboard, 70
Close, 40
Close button, 17
Communication port..., 48
Communications settings..., 42
CompuServe, 6
CompuServe B+, 29
Control characters, 56
Control Z, 44
Copy, 41
Copy to clipboard, 22
Copyright Notice, 1
Cut and paste, 20

D

Data region, 20
DB-25 port, 23
DCD, 73
Defective Disks, 3
Desk accessories, 33
Desk menu, 33
Dial menu, 57
Dialing, 27, 61
Dialog boxes, 12
Disabling STalker, 55
Disclaimer Notice, 3
Download binary file..., 37
Download directory, 53
Downloading, 28
DTR, 51, 73
Duplex, 25

E

Echo, 55
Edit menu, 41

F

FastLoad option, 74
Favour concurrency, 54
File menu, 34
File transfer, 28
File transfer settings..., 43
Flow control, 26
Full duplex, 25
Full size button, 17
Function key settings..., 49

G

GDOS, 48, 74
GEnie, 6
GMT, 53
Gribnif Software, 1, 5

H

Half duplex, 25
Hanging up, 28, 61
Hayes, 24
Help menu, 64
High speed modems, 24
Host prompt, 59

I

Installing STalker
- as a Desk Accessory, 8
- as a GEM Program, 9

Internet, 6

K

Keyboard shortcuts, 15

L

Licensing Agreement, 1

M

Master disk, 7
Menu bar, 14
- as a Desk Accessory, 15
- as a GEM Program, 14

Modem Settings..., 50
Modems, 23
MultiDesk, 74

N

NeoDesk, 69
NeoDesk Accessory, 69
Northampton, 5
Null modem cable, 23, 74

O

Open configuration file..., 34
Operating STalker, 12
Options menu, 54

P

Parity, 26
Paste, 41
Paste to STeno, 22
PC-ANSI, 26, 47
PinHead, 74
Preface, 1
Preference settings..., 52
Print, 22, 56
Problems, 5
Program overview, 4

Q

Quit, 40

R

READ_ME.NOW, 5, 32, 44
Receive text file..., 35
REGISTER.PRG, 7
Registering, 7
REMOTE.BTK, 39
Remote (mini-BBS) mode..., 38
Renting, 2
Reset communications port, 55. 72
Resize gadget, 18
RS-232, 51
RTS/CTS, 26
Running
- as a Desk Accessory, 11
- as a GEM Program, 11
- as a NeoDesk Accessory, 11

S

Save configuration, 34
Save to disk, 22
SCRAP.TXT, 70
Script directory, 53
Scroll bars, 19
Selecting a block, 21
Send text file..., 35
Setting X/Y/ZModem, 45
Settings menu, 42
Shadow access..., 40
Shutdown, 70
Software piracy, 1
Sounds, 54
STALKER.ACC, 8
STALKER.INF, 34
STALKER.PRG, 8
STALKER.RSC, 8
STalker info..., 33
Status line, 16
STeno, 37, 53, 65
Stop bits, 26
Support, 5

T

Terminal emulation, 26
Terminal settings, 27
Terminal settings..., 47
Text (ASCII) Files, 28
Text file download, 29
Text file upload, 29

Text settings, 43
Time stamps, 71
Title bar, 16
Transfer efficiency, 71
Transmit, 22
Type-ahead mode, 67

U

UNIX, 70
Upgrading STalker, 3
Upload binary file..., 36
Uploading, 28

V

VAX systems, 47
Virtual screen, 19, 20
VT-100, 26, 47
VT-52, 26, 47

W

Window controls, 15
Window style, 57
Word length, 25

X

XModem, 29, 31
XON/XOFF, 26

Y

YModem, 29, 31

Z

ZModem, 29, 31

BackTALK™

STalker 3's Advanced Scripting Language

GRIBNIF SOFTWARE

BackTALK

STalker 3's Advanced Scripting Language

Gribnif Software

STalker 3's **BackTALK**

Programs Copyright © 1988-1993 by Strata Software
Documentation & Packaging Copyright © 1991-1993 by Gribnif Software
All Rights Reserved

Enclusive World Wide Marketing and Distribution by Gribnif Software

STeno, STalker, and BackTALK are trademarks of Strata Software. NeoDesk is a registered trademark of Gribnif Software. All other trademarks belong to their respective holders.

No part of this book, disk, and/or software may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage or retrieval system, without the written permission of the publisher.

Program written and designed by Eric Rosenquist
Manual written by Bob Goff & Mike Cohan
Manual typeset by Rick Flashman
Manual edited by Dan Wilga & Rick Flashman

Gribnif Software
P.O. Box 779,
Northampton, MA 01061
U.S.A.

Tel: (413) 247-5620
Fax:(413) 247-5622

Printed in the United States of America
Second Edition

1. Table of Contents

1.. Introduction to BackTALK 1
1.1.. What is BackTALK? 1
1.1.1.. Creating BackTALK programs 1
1.1.2.. The BackTALK Compiler 1
1.2.. The STalker Compiled Script Interpreter 2

2.. Basic BackTALK Programming for Beginners 3
2.1.. Hello World 3
2.2.. Window Output: Printf 5
2.3.. Variables 5
2.3.1.. Types of variables 5
2.3.2.. Declaring variables 6
2.3.3.. Global and Local variables 6
2.3.4.. Assigning values to variables 7
2.4.. Printing Variables with Printf 8
2.5.. Input from the Keyboard 9
2.6.. Referencing Characters in Strings 10
2.7.. Flow Control: If-Then 10
2.8.. While-Endwhile & Repeat-Until 12
2.8.1.. WHILE-ENDWHILE 12
2.8.2.. REPEAT-UNTIL 13
2.8.3.. Difference between While and Repeat loops 14
2.9.. More About Functions 14
2.10.. Some Simple Modem Commands 16
2.11.. Summing up 17

3.. Advanced BackTALK Tutorial 18
3.1.. Getting Started 18
3.2.. A Stand-Alone Script 20
3.3.. Reading a String of Arbitrary Length 25
3.3.1.. More capabilities 30
3.4.. More Example Scripts 33
3.4.1.. BIXOFF.BTS 33
3.4.2.. FKEYS.BTS 34

4.. BackTALK Reference Manual 37
4.1.. Scripts 37
4.2.. Terms 38
4.3.. Data 39

4.3.1.. Integers 39
4.3.2.. Constants 40
4.3.3.. Strings 41
4.3.4.. String Pointers 42
4.3.5.. A note about strings 42
4.4.. Operators 43
4.5.. Functions 45
4.6.. Statements 45
4.6.1.. Assignments 45
4.6.2.. Comments 46
4.6.3.. EXIT Statement 46
4.6.4.. Function calls 46
4.6.5.. Function definitions 47
4.6.6.. Function declarations 47
4.6.7.. IF statement 48
4.6.8.. INCLUDE statement 48
4.6.9.. NULL Statement 49
4.6.10.. REPEAT statement 49
4.6.11.. RETURN Statement 49
4.6.12.. WHILE statement 50
4.7.. Built-In Functions 50
4.7.1.. A word about file names 50
4.7.2.. alert() 51
4.7.3.. appl_id() 51
4.7.4.. appl_find() 52
4.7.5.. appl_write() 52
4.7.6.. ascii_send() 53
4.7.7.. ascii_receive() 53
4.7.8.. atoi() 54
4.7.9.. chat_mode() 54
4.7.10.. crc() 54
4.7.11.. crc_update() 54
4.7.12.. crc32() 54
4.7.13.. crc32_update() 55
4.7.14.. dial_group() 55
4.7.15.. dial_load() 55
4.7.16.. dial_many() 55
4.7.17.. dial_one() 56
4.7.18.. dial_save() 56
4.7.19.. download() 56
4.7.20.. exit_program() 57
4.7.21.. file_attrib() 57
4.7.22.. file_change_dir() 58
4.7.23.. file_close() 58

4.7.24.. file_create() 58
4.7.25.. file_delete() 58
4.7.26.. file_dir_first() 59
4.7.27.. file_dir_next() 60
4.7.28.. file_exists() 61
4.7.29.. file_getchar() 61
4.7.30.. file_get_dir() 61
4.7.31.. file_get_drive() 61
4.7.32.. file_gets() 61
4.7.33.. file_instat() 62
4.7.34.. file_open() 62
4.7.35.. file_outstat() 63
4.7.36.. file_printf() 63
4.7.37.. file_putchar() 63
4.7.38.. file_puts() 64
4.7.39.. file_read() 64
4.7.40.. file_rename() 64
4.7.41.. file_seek() 65
4.7.42.. file_write() 65
4.7.43.. fkey_load() 65
4.7.44.. fkey_save() 66
4.7.45.. fsei() 66
4.7.46.. getchar() 66
4.7.47.. getkey() 67
4.7.48.. gets() 67
4.7.49.. get_baud() 68
4.7.50.. get_capture() 68
4.7.51.. get_duplex() 68
4.7.52.. get_emulation() 68
4.7.53.. get_flow_control() 68
4.7.54.. get_int_param() 69
4.7.55.. get_parity() 69
4.7.56.. get_port() 69
4.7.57.. get_printer() 70
4.7.58.. get_protocol() 70
4.7.59.. get_stop_bits() 70
4.7.60.. get_string_param() 70
4.7.61.. get_time() 71
4.7.62.. get_word_length() 72
4.7.63.. global_address() 72
4.7.64.. instat() 72
4.7.65.. isalnum() 72
4.7.66.. isalpha() 72
4.7.67.. iscntrl() 73

4.7.68.. isdigit() 73
4.7.69.. isgraph() 73
4.7.70.. islower() 73
4.7.71.. isodigit() 73
4.7.72.. isprint() 73
4.7.73.. ispunct() 73
4.7.74.. isspace() 74
4.7.75.. isupper() 74
4.7.76.. isxdigit() 74
4.7.77.. itoa() 74
4.7.78.. memory_read() 74
4.7.79.. memory_write() 74
4.7.80.. outstat() 75
4.7.81.. play_sound() 75
4.7.82.. port_break() 75
4.7.83.. port_carrier() 75
4.7.84.. port_connected() 75
4.7.85.. port_drop_dtr() 76
4.7.86.. prot_dtr() 76
4.7.87.. port_getchar() 76
4.7.88.. port_gets() 76
4.7.89.. port_hang_up() 77
4.7.90.. port_instat() 77
4.7.91.. port_outstat() 77
4.7.92.. port_printf() 77
4.7.93.. port_putchar() 77
4.7.94.. port_puts() 77
4.7.95.. port_send() 78
4.7.96.. port_write() 78
4.7.97.. printer_outstat() 78
4.7.98.. printer_printf() 78
4.7.99.. printer_putchar() 78
4.7.100.. printer_puts() 79
4.7.101.. printf() 79
4.7.102.. putchar() 80
4.7.103.. puts() 80
4.7.104.. run_program() 81
4.7.105.. set_baud() 81
4.7.106.. set_capture() 81
4.7.107.. set_duplex() 81
4.7.108.. set_emulation() 82
4.7.109.. set_flow_control() 82
4.7.110.. set_int_param() 82
4.7.111.. set_parity() 85

4.7.112.. set_string_param() 85
4.7.113.. set_port() 87
4.7.114.. set_printer() 88
4.7.115.. set_protocol() 88
4.7.116.. set_stop_bits() 88
4.7.117.. set_time() 89
4.7.118.. set_word_length() 89
4.7.119.. show_message() 89
4.7.120.. sizeof() 89
4.7.121.. sprintf() 89
4.7.122.. steno_append() 90
4.7.123.. steno_id() 90
4.7.124.. strcat() 90
4.7.125.. strchr() 90
4.7.126.. strcmp() 91
4.7.127.. strcpy() 91
4.7.128.. strftime() 91
4.7.129.. strlen() 92
4.7.130.. strlwr() 92
4.7.131.. strncat() 92
4.7.132.. strncmp() 92
4.7.133.. strncpy() 93
4.7.134.. strrchr() 93
4.7.135.. strstr() 93
4.7.136.. strtoi() 93
4.7.137.. strupr() 94
4.7.138.. timer_check() 94
4.7.139.. timer_elapsed() 94
4.7.140.. timer_set() 94
4.7.141.. tolower() 95
4.7.142.. toupper() 95
4.7.143.. upload() 95
4.7.144.. wait() 95
4.7.145.. wait_for() 95
4.7.146.. wait_until() 96
4.7.147.. write_script() 96

5.. Appendix A - ASCII Table 97

6.. Appendix B - Keyboard Scancodes 102

7.. Appendix C
- Source Code Formatting Conventions 103

8.. Appendix D - BackTALK Operators 105
8.1.. Binary Operators 105
8.2.. Unary Operators 105
8.3.. Operator Precedence 106

9.. Appendix E - The BackTALK Compiler 107

10.. Appendix F
- BackTALK Compiler Error Messages 108

11.. Appendix G - File Operation Error Codes 111
11.1.. BIOS Errors 111
11.2.. GEMDOS Errors 111
11.3.. BackTALK Errors 112

12.. Appendix H - Emulations 113
12.1.. Keyboard Generated Sequences 114
12.1.1.. BREAK 114
12.1.2.. Carriage Return 114
12.1.3.. Cursor Keys 114
12.1.4.. Keypad 114
12.2.. Received Sequences 115
12.3.. VT100 Control Codes 116
12.4.. VT100 Mode Setting Sequences 116
12.5.. VT100 Control Sequences 116
12.6.. VT100 Video Attributes 119
12.7.. VT100 Character Set Selection 120
12.8.. VT52 Escape Sequences 120

13.. Index 122

1. Introduction to BackTALK

1.1. What is BackTALK?

BackTALK is the scripting language supplied with STalker. With BackTALK you can write programs to automatically do anything you can do with STalker. You can automate those repetitive telecommunications tasks like logging in every day to check your mail, setting up your host system's parameters, or anything else you can think of.

BackTALK is a structured, block oriented, compiled programming language inspired by the 'C' programming language. Although the syntax is slightly different, the operators and control structure are similar. If you have any familiarity with C or Pascal, you should have no trouble learning BackTALK if you read the *Advanced Programming Tutorial*, or even just the *BackTALK Reference Manual.* If not, don't worry; we've included an introductory tutorial as well.

1.1.1. Creating BackTALK programs

To create BackTALK programs, you will need a text editor capable of writing ASCII (text) files. *STeno* from Gribnif Software is ideal for this purpose. If you are using a regular word processor, be sure to save your program in ASCII format.

1.1.2. The BackTALK Compiler

The BackTALK compiler takes the file containing your BackTALK program, and from it generates a new file in a format which the *STalker Compiled Script Interpreter* can understand. This process is called compiling, and the file created by the compiler is called a compiled script. If there are any syntax errors in your program which prevent it from being compiled, the compiler will display them. Once you have the compiled script, you can execute it by using the *Run BackTALK script...* command from *STalker's BackTALK* menu.

The BackTALK compiler, may be run either as a stand-alone program or as a desk accessory. To run it as a stand-alone program, name the file BACKTALK.PRG and double-click on it. To use it as a desk accessory, name the file BACKTALK.ACC, place it in the same directory from which your other accessories load, and reboot (reset) your computer. You will see the compiler in the Desk or (Atari Fuji) drop-down menu.

When you run the compiler, it displays a quit button and a button labeled *Compile .BTS File...* The latter displays the GEM file selector, allowing you to choose the file to be compiled. BackTALK program files should be in ASCII format (all text) and have a file extension of ".BTS". The *BackTALK Compiler* creates scripts with an extension of ".BTK".

The center of the compiler's window is a status field, showing you which lines of your program had errors and what the error message was. If there were no errors, you will see the file name of the compiled script, and the message "BackTALK file written successfully!"

1.2. The STalker Compiled Script Interpreter

The STalker compiled script interpreter is invoked within STalker by using the *Run BackTALK Script* option from the BackTALK drop-down menu. The GEM file selector appears, allowing you to choose the .BTK script to execute.

2. Basic BackTALK Programming for Beginners

This chapter will not attempt to explain every command and element in the BackTALK language. Rather, it is an introductory tutorial in programming in BackTALK, using some of the most basic commands and elements. If you are familiar with programming in C or Pascal, you may want to skip to the *Advanced BackTALK Programming Tutorial.*

2.1. Hello World

The very first program everybody writes is the "Hello World" program. This program simply writes the words "Hello World" to the screen. Here is this program in BackTALK:

```
Function main()
   Printf("Hello World!\r\n");
Endfunction
```

To see this program work, first create the file with your text editor. Then, save it to disk with a filename of HELLO.BTS. Next, invoke the BackTALK compiler by either running the program or calling the desk accessory. Choose the Compile .BTS File button. When the GEM file selector pops up, select the file you just created. After a brief pause, you should see the message BackTALK file written successfully! If you get any error messages, go back and edit your file to remove the mistakes, then recompile. If your file was OK, there will now be a new file in the same directory you saved HELLO.BTS to. This file will be called HELLO.BTK, and is the compiled script. Run STalker, choose *Run BackTALK Script...* from the BackTALK menu, and choose HELLO.BTK from the GEM file selector. If you did everything right, you should see the words:

```
Hello World!
```

displayed in the STalker window.

This program shows you the basic minimum structure of a BackTALK program. Every BackTALK program contains at least one function. A function is a group of

programming commands that, taken together, do something (in this case, print "Hello World!" to the screen).

The *main function*, indicated by the line `Function main()`, is contained within every BackTALK program. This is always where execution (operation) of the program begins. The line `Endfunction` is associated with `Function main()` and indicates where the `Main` function ends. In between the name of the function and the `Endfunction` are the lines of proram code that make the function do something.

Function names you create are case sensitive; that is, `Function main()` is not the same thing as `Function Main()`. If you get strange errors while compiling, check to make sure that your upper and lower case letters are consistent each time you refer to a function. Built-in function names are not case sensitive, however.

Programs can have more than one function. To illustrate this, try this program (call it HELLO2.BTS) :

```
Function hello()
   Printf("Hello World!\r\n");
Endfunction

Function main()
   hello();
   Printf("Now we are done!\r\n");
Endfunction
```

When you execute this script, you will find that it displays

```
Hello World!
Now we are done!
```

in the STalker window.

You can see that this program has two functions, `hello` and `main`. Each function starts with a `Function name()` line, where *name* is the name of the function, and ends with an `Endfunction` line.

Let's trace our way through the operation of this program. Remember that execution of the program always begins at the `Function main()` line, so that is where we start. The next line, `hello();` is the name of a function previously defined in the program. This tells BackTALK to call that function (that is, jump to the beginning of that function and execute each line until it hits an `Endfunction`, then return to the previous point.) Therefore, when we hit the `hello();` line, BackTALK jumps to the line `Function hello()`. The next line, `Printf("Hello World!\r\n");` displays the words `Hello World!` in the STalker window. (Exactly how the `Printf` command works will be explained later.) The next line is `Endfunction`, which tells BackTALK that the `hello` function is over. Therefore, BackTALK jumps to the next line after the one that called the function. This line is `Printf("Now we are done!\r\n");`

After this line executes, the next line is `Endfunction` which marks the end of the program.

Try your hand at writing some BackTALK programs using multiple functions and the `Printf` command. Note that `Printf` lines end with a semicolon, but `Function` and `Endfunction` lines do not.

2.2. Window Output: Printf

As you have already seen, the `Printf` command outputs information in the STalker window. If you are only displaying some text, the format of the `Printf` command is `Printf("text");` where text is the information you want displayed. If you end the text with `\r\n` (`\r` is a carriage return, `\n` is a linefeed), the cursor will move to a new line after your text displays.

```
Printf("This is a test! ");
Printf("This is a test! ");
```

displays:

```
This is a test! This is a test!
```

and leaves the cursor at the end of the line.

```
Printf("This is a test!\r\n");
Printf("This is a test!\r\n");
```

displays:

```
This is a test!
This is a test!
```

and leaves the cursor on a new, blank line.

2.3. Variables

2.3.1. Types of variables

Variables let your program "remember" and act on information. Each variable has a name, a type, and a value. The name is constant after you define it, and is what you use within your program to refer to the variable. The value of the variable changes as your program works with it. When you tell your program to output a variable to the STalker screen, the value of the variable is what appears, not the name. Variable names are case sensitive, so always remember to keep your capital and small letters consistent when you use variables.

There are two kinds of variables in BackTALK: strings and integers. String variables hold text (the text is called a string), and integer variables hold integers (whole numbers).

2.3.2. Declaring variables

Before you can use a variable, you must *declare* the variable. Declaring a variable assigns it a name and tells BackTALK what type of variable it is.

Integer variable declarations have the format *Int name*; where *name* is the name of your variable. For instance, `Int X;` would define a variable called X.

String variable declarations have the format *String name[size]*; where *name* is the name of the variable and *size* is a number specifying the maximum number of characters of text the variable can hold. One character at the end of the string is invisible and is used to mark the end of the string, so if you used a string definition of `String letters[10]`, the variable `letters` would actually be able to hold 9 characters of text, not 10. You can leave the size declaration, *[size]*, off. If so, you have declared a *string pointer*. The difference between a string and a string pointer is somewhat abstract. See the *BackTALK Reference Section* for a detailed description.

2.3.3. Global and Local variables

You can declare a variable outside a function, like this program does:

SAMPLE PROGRAM C-1

```
Int X;
Function something()
    (stuff for function)
Endfunction

Function somethingelse()
    (stuff for function)
Endfunction

Function main()
    something();
    somethingelse();
Endfunction
```

If your variable is defined outside a function, the variable may be used by any function in your program, as long as the function comes after the variable declaration. Therefore, in sample program C-1, if function `something` put a value in X, function `somethingelse` could then use X and the value in it. Variables which are defined outside a function are called *global variables*, because they are available globally (that is, anywhere in the program).

You can also declare a variable inside a function, like this program does:

SAMPLE PROGRAM C-2

```
Function something()
    Int X;
    (stuff for function)
Endfunction

Function somethingelse()
    (stuff for function)
Endfunction

Function main()
    something();
    somethingelse();
Endfunction
```

Sample program C-2 looks very similar to program C-1, but there is an important difference. In C-2, the variable X is declared inside function `something`, and so it will only exist while function `something` is being executed. As soon as function `something`'s `Endfunction` is reached, variable X disappears. If function `somethingelse` tries to use X, BackTALK will generate an error. Variables of this type are called local variables, because they are local to the function that created them.

2.3.4. Assigning values to variables

Once you have defined a variable, you will want to assign a value to it. Doing this within your program is easy; just put the variable name, an equals sign (=) and the value you want.

If you defined a string with a length, you can only assign values to each character of the string individually. For instance, if you defined a string like this: `string st[10];`, you have 9 characters to work with (the 10th is used to mark the end of the string). To assign a value to each character of this string, use the format `st[x]='a'`; where x is a number from 0 to 8, 0 being the first character and 8 being the last, and a is the character you want.

If you defined a string pointer, you can assign a string to it all at once by enclosing the text in double quotes, like this:

```
st="testing!";
```

Note: A single character is always enclosed in single quotes, like `'a'`. A string is enclosed in double-quotes.

Here is a sample program showing some variable declarations and assignments. Notice that st and ST are actually different variables because the case used in their names is different.

SAMPLE PROGRAM C-3

```
Int x;
String st;
String ST[10];

Function main()
   x=5;
   st="text";
   ST[0]='t';
   ST[1]='e';
   ST[2]='x';
   ST[3]='t';
endfunction
```

2.4. Printing Variables with Printf

Using variables with the `Printf` command can get a little tricky. Recall that if you execute the following line:

```
Printf("Hello!\r\n");
```

you get the word `Hello!` on the screen and then the cursor moves to a new line. But what if you have variables? How do you know if the letter X is in fact the letter or a variable name? For this reason, you cannot just mix variables into your text. You need a way to differentiate between the two, and here is how it works. Take sample program C-3 and add the following line to it, after the line `ST[3]='t';`

```
Printf("x is %d and st is %s and ST[0] is %c\r\n",x,st,ST[0]);
```

Recompile and run the program, and you will get the following output:

```
x is 5 and st is text and ST[0] is t
```

Now, examine the `Printf` line and the resulting output carefully. Notice that the value of the variable `x` (which is 5) is being substituted for the characters `%d`, the value of the variable `st` (which is 'test') is being substituted for the characters `%s`, and the value of the variable `ST[0]` (which is 't') is being substituted for the characters `%c`. (As before, `\r\n` moves the cursor to a new line.)

The `%d` is a code indicating that the first variable in the list following the quotation marks should be printed here ('d' indicates a numeric value), `%s` is a code indicating that the second variable in the list should be printed here ('s' indicates a string), and `%c` is a

code indicating that the third variable in the list goes here (%c is an individual character.) So, you can print as many variables as you want by simply listing them after the quotes, and putting the appropriate codes in your text.

There are many additional ways to format variable output in a `Printf` command. See the *BackTALK Reference Section* for details.

SAMPLE PROGRAM C-4

```
Int x;
Int y;
Int z;

Function main()
   x=1;
   y=2;
   z=x+y;
   printf("%d+%d=%d\r\n",x,y,z);
endfunction
```

This program produces the output:

```
1+2=3
```

Examine it and make sure you understand why it works. Try changing some of the values and recompiling. How would you modify the `printf` line to produce 3=1+2 instead?

2.5. Input from the Keyboard

The command `Gets` (short for get string) lets you receive input form the keyboard. It has the format *length=Gets(string, max_length)*. *String* is the variable which holds whatever the user typed in, *max_length* is a number (or an integer variable) indicating one more than the maximum length of *string*, and *length* is a variable holding the number of characters typed.

Here is a sample program using *Gets*:

```
String st[20];
int numchars;

Function main()
   printf("Type something in, then press Return! >>");
   numchars=gets(st,10);
   printf("You typed %d characters.\r\n",numchars);
   printf("You typed %s.\r\n",st);
endfunction
```

This program tells you how many characters you typed, then shows you what you typed.

2.6. Referencing Characters in Strings

Each character in a string is numbered, in order from first to last. The total list of characters is called an array, and the numbering runs from 0 to (number of characters - 1). To reference an individual character, the format is: *stringname[x]*, where *x* is the number of the character you want to use. Remember that since the numbering starts at 0, 0 references the first character, 1 the second, and so on. Examine this sample program:

```
String st[15];
int numchars;
function main()
    printf("Type in at least 3 characters, then press Return! >>");
    numchars=gets(st,10);
    printf("The third character you typed was %c\r\n",st[2]);
endfunction
```

This program prints the third character you enter. Notice the use of %c. This prints a single character.

2.7. Flow Control: If-Then

Now that you know how to get input from the keyboard, you need to know how to act on that input. For instance, you might want to have your script say "Type A to do this and B to do that", and then execute function A if the user typed A and function B if they typed B. To accomplish this, you can use the IF-THEN-ELSE-ENDIF command. Here is a sample program:

SAMPLE PROGRAM E-1

```
String st[10];
int numchars;

Function main()
    printf("Type either 1 or 2. >>");
    numchars=gets(st,5);
    IF (st[0]=='1') THEN
        printf("You typed a 1!\r\n");
    ELSE
        printf("You typed a 2!\r\n");
    ENDIF
endfunction
```

The IF-THEN statement evaluates the logical statement following the 'IF'. If the statement evaluates to 'TRUE', then the program section following the THEN is

executed, up until the ELSE. If the statement evaluates to 'FALSE', then the statement following the ELSE is executed, up until the ENDIF. So in the sample program, the statement (ST[0]=='1') is tested. If ST[0] (the first character you entered) is in fact equal (==) to the character 1 ('1'), then the statement evaluates to TRUE, and the line `printf("You typed a 1!\r\n");` is executed. If ST[0] is not 1, then the statement evaluates to FALSE, and the line following the ELSE, `printf("You typed a 2!\r\n);` is executed.

If you don't want to use an ELSE section, you can leave it out. Just end the IF-THEN with an ENDIF.

Note the use of the double equal sign in the logical statement. When you assign values to variables, like "X=1;", you use a single equal sign. The double equal sign is only used in logical expressions. Some other logical signs are: > (greater than), >= (greater than or equal), < (less than), <= (less than or equal), and != (not equal to).

If you study sample program E-1 carefully, you will notice that there is a 'bug' in it. That is, there is a logic error. The error is that if you type anything other than 1, the program prints "You typed a 2!", even if you typed something else. To fix this problem, we can use nested IF-THEN statements. This simply means that the ELSE section of the IF-THEN is itself an IF-THEN. The following program fixes the bug:

SAMPLE PROGRAM E-2

```
String st[10];
int numchars;

Function main()
   printf("Type either 1 or 2. >>");
   numchars=gets(st,5);
   IF (st[0]=='1') THEN
      printf("You typed a 1!\r\n");
   ELSEIF (st[0]=='2') THEN
      printf("You typed a 2!\r\n");
   ELSE
      printf("Can't you read?\r\n");
   ENDIF
endfunction
```

Observe that the program first tests `st[0]` to see if it is a '1'. If not, it then executes the ELSEIF. This tests `st[0]` to see if it is a '2'.

2.8. While-Endwhile & Repeat-Until

2.8.1. WHILE-ENDWHILE

WHILE-ENDWHILE and REPEAT-UNTIL are structures that let you repeat specified sections of your program until some condition is met. For instance, suppose you wanted to force the user of your program to type a "Y" or an "N". You would want to keep looping around, executing Gets calls and checking what the user typed in, until a "Y" or "N" is entered. Try this example using WHILE-ENDWHILE:

SAMPLE PROGRAM F-1

```
String st[10];
int ok;
int numchars;

Function main()
   ok=0;
   WHILE (ok==0) DO
      printf("Type either Y or N. >>");
      numchars=gets(st,5);
      IF (st[0]=='Y') THEN
         ok=1;
      ELSE
         IF (st[0]=='N') THEN
            ok=1;
         ENDIF
      ENDIF
   ENDWHILE
   printf("You typed a %c.\r\n",st[0]);
endfunction
```

In this program, the variable ok is being used as a 'flag', that is, it 'flags' when to exit the loop. The While loop is marked at the beginning by the WHILE (logical statement) DO, and at the end by an ENDWHILE. Everything in between the WHILE-DO and ENDWHILE is called the 'body of the loop'. The body of the loop is the part that gets repeated.

A WHILE-DO-ENDWHILE loop is executed as follows. When the program first encounters the WHILE (*logical statement*) DO line, it tests the logical statement to see if it is true or false. If the statement is true, the body of the loop is executed, and when the program gets to the ENDWHILE, it jumps back to the WHILE-DO line and starts over. If the statement is false, the loop is skipped, and the program jumps to the next line following the ENDWHILE and continues executing.

Let's see how this works in our sample program. When the program first comes to the line `WHILE (ok==0) DO`, it tests the statement (ok==0) to see if it is true or false.

In this case, it is true, because we set ok to 0 in the previous line. Therefore, the program executes the body of the loop. When the program reaches the end of the loop, marked by the ENDWHILE statement, it jumps back to the beginning of the loop and starts over. Now, look at the body of the loop. First, the user types something in, and that text is stored in variable st. Now, look at the two nested IF-THEN statements. They test the first character of st. If that character is a 'Y' or an 'N', then the value of variable ok is changed to 1. So, when we first start the loop, ok is equal to 0, but when we jump back to the beginning of the loop, if the user typed a Y or an N, ok will then be equal to 1. Since the statement (ok==0) evaluates to FALSE if ok is 1, as soon as the user types a Y or an N, the loop terminates and the program jumps to the line following the ENDWHILE, which is the printf statement. As long as the user does not type a Y or N, ok stays 0 and the loop continues.

2.8.2. REPEAT-UNTIL

The other kind of loop is the REPEAT-UNTIL loop. Here is a program that does the same thing as program F-1, but using REPEAT-UNTIL.

SAMPLE PROGRAM F-2

```
String st[10];
int ok;
int numchars;

Function main()
   ok=0;
   REPEAT
      printf("Type either Y or N. >>");
      numchars=gets(st,5);
      IF (st[0]=='Y') THEN
         ok=1;
      ELSE
         IF (st[0]=='N') THEN
            ok=1;
         ENDIF
      ENDIF
   UNTIL (ok==1)
   printf("You typed a %c.\r\n",st[0]);
endfunction
```

This program is very similar to program F-1. The Repeat loop is marked at the beginning by the REPEAT line, and at the end by the UNTIL (logical statement) line. When the program hits the REPEAT line, it executes the body of the loop, until it hits UNTIL. Then, it tests the logical statement following the UNTIL. If the statement evaluates to FALSE, the program jumps back to the beginning of the body of the loop and starts over. If it evaluates to TRUE, the program continues with the next line.

2.8.3. Difference between While and Repeat loops

You can see that REPEAT-UNTIL and WHILE-DO-ENDWHILE are very similar. If you are really catching on, you have probably noticed that in a WHILE-DO loop, the loop continues as long as the test condition is TRUE, while in a REPEAT-UNTIL loop, the loop continues as long as the test condition is FALSE. There is another, more important difference between the two kinds of loops, and it is this difference which determines which type you will use in a given situation.

In a WHILE-DO loop, the condition is tested at the top of the loop. If the condition is FALSE, the loop is skipped. Therefore, in a WHILE-DO loop, it is possible for the body of the loop to be skipped entirely if the test condition is FALSE when the loop is first encountered.

In a REPEAT-UNTIL loop, however, the condition is tested at the bottom of the loop. This means that the body of the loop is always executed at least once.

2.9. More About Functions

We discussed functions a little at the beginning of this tutorial. You will recall that functions begin with a `function` *name*`()` section, where *name* is the name of the function, and end with an `Endfunction`. After a function is defined, it can be invoked anywhere in your program by using the function name.

Functions are used to break your program up into readable parts that can be executed a number of times. Each function should be given a descriptive name that describes what the function does.

You may or may not have realized that some of the capabilities of BackTALK such as `Printf` and `Gets` are, in fact, functions. They are simply functions that you do not have to define because they are built into BackTALK. This brings us to the next subject, function parameters.

Remember that each function name is followed by a set of parentheses. In functions that do not use parameters, these parentheses do not contain anything. In functions that do use parameters, these parenthesis contain a list of variables or data. This data is passed to the function, so that the function can use and/or modify the data. For instance, when you say `Printf("yeeha!");`, the "yeeha!" is a parameter that is being passed to the `Printf` function.

To define a function that uses parameters, put the type and variable name inside the parentheses. For example, if you want to define a function that uses one string and one integer variable, you can say: `Function demo(string st, int number);`. The only restriction is that strings in parameter lists must not have a length (they must be string pointers.) Use unique names for your parameter variables; do not duplicate the names of global variables, even though you can do it.

When you call the function, put the variables and/or values you want the function to use inside the parentheses, like this: `demo("xyzzy",4);` or `demo(st,x);`.

Try this example to see some simple parameter passing:

```
int x;
string st[10];

function demo(string ST, int y);
   printf("ST is %s, y is %d\r\n",ST,y);
endfunction

function main()
   st[0]='a';
   st[1]='b';
   st[2]='c';
   x=5;
   demo (st,x);
endfunction
```

Functions can also return a value. You have already seen this in the built-in `gets` function. Recall that the format of this function is `length=gets(`*String*`,`*Maxchars*`);`. *String* and *Maxchars* are parameters. *Length* is a value returned by the function, namely, the number of characters in the variable *String*.

To make a function return a value, add `returns int` or `returns string` to the end of the function's name, as in `function demo (int x) returns int;`. As an example, here is a simple function that takes an integer parameter and returns that number plus 1.

```
Function addone (int z) returns int
   return z+1;
endfunction
```

The `return` line exits the function and jumps back to the location in the program where the function was called, returning whatever value you give. Here is this same function, used in a program:

```
int x;
int y;

function addone (int z) returns int
   return z+1;
endfunction
```

```
function main()
   x=addone (5);
   printf("5+1 is %d\r\n",x);
   y=addone (x);
   printf("5+1+1 is %d\r\n",y);
endfunction
```

As you can see from this example, a function that returns a value is used in an expression just as if it were a variable. Just as you could say "y=5", you can say "y=addone (x);", because addone returns an integer. Another thing you may notice by looking at this example is that you can pass variables like x to a function or you can pass constants like 5.

2.10. Some Simple Modem Commands

The port_printf command and the port_gets functions work similarly to the regular printf and gets functions, but they send or receive information from the modem rather than the screen and keyboard.

The port_printf function has an identical format to printf, but information will be sent over the currently selected port, using the current settings. There are BackTALK functions to change these settings. See the *BackTALK Reference Manual* for details.

The port_gets command has the format port_gets(st,num_chars); where st is of type string and *num_chars* is of type int. The function waits to receive data from the modem. It returns either when it has received num_chars characters, or when it receives a linefeed. The data received is stored in st.

Try the following sample program if you have a Hayes-compatible modem. Before running it, turn the modem on.

```
String info[10];

function main()
   port_printf("ATZ\r");
   port_gets(info,10);
   printf("Modem returned: %s",info);
   port_gets(info,10);
   printf("Modem returned: %s",info);
endfunction
```

When you run this program, STalker will display:

```
Modem returned: ATZ
Modem returned: OK
```

The BackTALK program used the `port_printf` function to send the letters ATZ followed by a carriage return to the modem, which echoed them back with a linefeed after the carriage return. Thus, the first line received by STalker was ATZ [return-linefeed], which was stored in the Atari's serial port input buffer. Next, the modem, having received the ATZ command, responded with OK [return-linefeed]. This line was also stored in the Atari's serial port input buffer. The program then executed the `port_gets` function, which read the first line from the buffer. This line was ATZ, which was displayed by the first `printf` call. Next, the BackTALK program executed a second `port_gets`, which read the next line in the buffer: OK. This line was displayed by the second `printf` command.

2.11. Summing up

Having read the preceding chapters, you should now try your hand at writing programs in BackTALK. Don't try anything too complicated at first; try to learn the syntax and figure out common errors.

Once you master the basics, you will want to learn to use BackTALK for its real purpose: interfacing with STalker to automate your telecommunications. BackTALK has dozens of functions allowing you to control every aspect of STalker. To help you understand these functions by seeing them in actual programs, there is a folder on the STalker disk called SCRIPTS containing sample scripts. These scripts incorporate many of the functions previously shown, as well as many of the STalker control functions. Once you understand the programming concepts, you can read the *Advanced BackTALK Tutorial* for a detailed walk-through in programming in BackTALK for a real application.

3. Advanced BackTALK Tutorial

3.1. Getting Started

We're going to start out here with a simple BackTALK script to log on to an electronic service. Let's assume that it will be connected to an autodialer entry as an autoexecute script, so that it takes over as soon as the connection is made.

```
FUNCTION main()
   wait_for("Name?", 0);
   port_puts("myname\r");
   wait_for("word:", 0);
   port_puts("myword\r");
ENDFUNCTION
```

Script execution starts with the `main()` function. We use the built-in function `wait_for()` to pause the script until the desired prompt appears, then `port_puts()` to send the appropriate response. Since `wait_for()` is one of the functions that allow incoming data to appear on the screen, we still see everything that goes on. This is important, since only while the script is in a waiting mode is incoming text sent to the terminal window. The rest of the time, the script itself is responsible for taking data out of the communications buffer and processing it. This avoids the problems of timing that can be so variable. Your script doesn't have to 'catch' the data just right, it can process all the data as it comes in.

Note that the responses include a return character "`\r`". None of the built-in functions automatically append a return. If you would normally type in some characters and press the [Return] key, you should add a `\r` character to the end of the string in BackTALK. Also remember to include the timeout value in the `wait_for()` function. Not including all the parameters that a function needs will cause strange malfunctions that are often difficult to track down.

This is a very simple script which does the same thing as putting the appropriate prompts and responses into the logon sequences for the autodialer entry. However, while this is about as much as the logon sequences can do, the script is just getting started.

Lets say that this script is for logging onto BIX (Byte Information eXchange) via Tymnet, a data network. Since this is a relatively complex process, it will be a good exercise.

When first dialing the local Tymnet number, most people get a line of garbled characters on their screen. This is really the prompt 'please enter your terminal identifier' sent at 300 bps. The response is always "a" without a return. Once Tymnet has your terminal identifier, it responds with a readable prompt "please log in:". To get to BIX from there, enter "bix" with a return. What will that look like in our script?

```
FUNCTION main()
STRING   log_id;
STRING   pword;
   log_id  = "myname\r";
   pword   = "myword\r";
   REPEAT
      port_putchar('a');
   UNTIL wait_for("log in:", 5);
   REPEAT
      port_puts("bix\r");
   UNTIL wait_for("call connected", 50);
   wait_for("Name?", 0);
   port_puts(log_id);
   wait_for("word:", 0);
   port_puts(pword);
ENDFUNCTION
```

First, we've declared two string pointers named log_id and pword. Then we assign them to point to two string literals, "myname\r" and "myword\r". This is not the same thing as declaring two strings and putting those contents in them, although for now the difference isn't important. We did that because it makes things easier as the script gets more complicated; we may want to use those pointers in more than one place. This way, when they need to be changed, we only have to change them once.

The next addition is a REPEAT statement. Inside the REPEAT statement, a port_putchar() function sends a single character, "a", to the communications port. Then the REPEAT statement checks the return value of a wait_for() function with a .5 second timeout. The wait_for() is waiting for the string "log in:", and returns FALSE if it doesn't get it within the timeout period. When it does get it, wait_for() returns TRUE, the REPEAT statement is satisfied, and the script continues. The effect of this is to send the character "a" to the communications port every half second until the "please log in:" prompt appears.

Note that we don't worry about timing here. The .5 second timeout in the wait_for() could just as well be .1 second or 5 seconds, the script would still work because all the incoming data is held for the wait_for() function to process.

A similar REPEAT statement sends "bix\r" every 5 seconds until the "BIX: call connected" message appears. This message is confirmation from Tymnet that

we are finally connected with BIX. Then our original `wait_for()` and `port_puts()` response pairs get us logged onto BIX.

3.2. A Stand-Alone Script

As we said above, this script assumes that it takes over from the autodialer as soon as the connection is made. What if we wanted to make this a stand-alone script, that might, for example, be called from a function key? We can make the same script do both jobs by using the `port_connected()` function.

```
FUNCTION main()
STRING   log_id;
STRING   pword;
   log_id  = "myname\r";
   pword   = "myword\r";
   IF not port_connected() THEN
      puts("\r\nDialing Bix+\r\n");
      IF not dial_one("bix") THEN
         exit 1;
      ENDIF
   ENDIF
   REPEAT
      port_putchar('a');
   UNTIL wait_for("log in:", 5);
   REPEAT
      port_puts("bix\r");
   UNTIL wait_for("call connected", 50);
   wait_for("Name?", 0);
   port_puts(log_id);
   wait_for("word:", 0);
   port_puts(pword);
ENDFUNCTION
```

Here we've added an IF statement at the top of the script. It checks the return from a `port_connected()` function call, which returns TRUE if STalker has made a connection with the autodialer. If the `port_connected()` function returns false, a message is printed to the terminal window with the `puts()` function saying that the script is dialing, then the autodialer entry "`bix`" is called with the `dial_one()` function. Another IF statement checks the success of the dialing operation and stops the script with a status of 1 (indicating a general error) if a connection was not made. Once the `dial_one()` function returns successfully, the rest of the script proceeds as before. Now, if the script is called outside the autodialer it will use the autodialer to dial the modem and set the port and terminal parameters and then log on.

Note that we used the `dial_one()` function in preference to dialing the modem directly from the script using a `port_puts()` call. For one thing, this allows us to keep

the phone number in the autodialer where it is more easily changed than in a compiled script. Also, the autodialer automatically sets the port and terminal parameters appropriately. The script could be made to do that, but allowing the autodialer to do it is much simpler and easier to maintain.

Speaking of function keys, we can make the logon script automatically set up the same function key to log off every system we log onto. If we are connected to BIX via Tymnet, to log off we enter "bye" with a return, wait for the "please log in:" prompt, and enter [Control][Z] to tell Tymnet to hang up. The function key to do that might look like this:

```
bye C/R 1/2 1/2 1/2 1/2 [Control][Z]
```
(where C/R is [Control][M] and 1/2 is the [Insert] key's delay code)

This sends "bye" with a return, waits two seconds, then sends [Control][Z].

```
FUNCTION main()
STRING   log_id;
STRING   pword;
STRING   bye_string;
   log_id  = "myname\r";
   pword   = "myword\r";
   bye_string = "bye\r\xAB\xAB\xAB\xAB\x1A"
   IF not port_connected() THEN
      puts("\r\nDialing Bix+\r\n");
      IF dial_one("bix") THEN
         exit 1;
      ENDIF
   ENDIF
   REPEAT
      port_putchar('a');
   UNTIL wait_for("log in:", 5);
   REPEAT
      port_puts("bix\r");
   UNTIL wait_for("call connected", 50);
   wait_for("Name?", 0);
   port_puts(log_id);
   wait_for("word:", 0);
   port_puts(pword);
   set_string_param(FKEY_PARAM, 9, bye_string);
ENDFUNCTION
```

Here we've added another string pointer, bye_string, and assigned it to point to the string we want to put in our function key. Then, down at the bottom of the script, after successfully logging on, we use the set_string_param() function to put the contents of bye_string into F10.

Since most text editors will balk at entering special characters like a carriage return and [Control][Z] in the middle of the string, constant notation has been used to enter them. The "\" character in a string tells the BackTALK compiler that special information follows. "\r" indicates a carriage return, ASCII 13. The "\xAB" is interpreted as a hexadecimal number, in this case the Atari code for the "½" character, and the "\x1A" is interpreted as ASCII 26, or [Control][Z].

An alternative to putting the logoff string in the function key itself is to make the function key call another BackTALK script using the special character *f* (ASCII 0xDD), which in turn sends the logoff string. The advantage is that the script can do more than just send the string. It might even restore the F10 key to its former state. What if the logon script had saved the contents of the function keys before it changed the F10 key? Then the logoff script could simply restore them as they were before. Let's write a pair of companion functions to save the contents of the function keys to a disk file and restore them. BackTALK has FKEY_Load and FKEY_Save functions that are similiar to these but use the same file format as interactively picking Load/Save from the FKey dialog box.

```
////////////////////////////////////////////
//  save_fkeys();
//
//  Syntax: status = save_fkeys(STRING file_name);
//
//  Saves the contents of the STalker function key strings in the
//  file. The file is in the format "key_number,
//  key_string_length, key_string". Returns 0 if successful, or
//  the negative error code.
////////////////////////////////////////////
FUNCTION save_fkeys(string file_name) RETURNS INT
INT handle;
INT key;
STRING key_buffer[41];
key=0;
IF (handle=file_create(file_name)) < 0 THEN
   return handle;            // file creation error
ENDIF
// store the key number, string size, string for 20 keys
REPEAT
   get_string_param(FKEY_PARAM, key, key_buffer, sizeof(key_buffer));
   file_putchar(handle, key);
   file_putchar(handle, strlen(key_buffer));
   file_puts(key_buffer, handle);
   key=key+1;
UNTIL key == 20;
// close up, return success
file_close(handle);
return 0;
ENDFUNCTION
```

Our function save_fkeys() begins by declaring integer variables for the file handle and the function key number, and a string to hold the contents of the function key. The buffer is dimensioned to hold 41 bytes, 40 possible characters in the function key string and one trailing null. It initializes the function key number to 0, then tries to create the file to store the function key text in. If the handle returned from

file_create() is negative, an error has occurred. In that case, the handle is the error code, which is returned to whatever function called save_fkeys() for proper handling.

Inside the REPEAT statement, we then use get_string_param() to copy the contents of each function key into the string key_buffer. The function key number and the number of characters in the string are stored in the file using file_putchar(), followed by the string itself using file_puts(). We store the length of the string to simplify retrieval. Since the strings themselves can contain return and line feed characters, we have to rely on some other means of separating them within the file.

Note that we could do just as well by omitting the function key number. Since we're storing all the function key strings in order, it's simple to retrieve them into the right function keys. This format, however, will allow for some future function to save and restore partial sets of function keys and use the same files created by these functions.

In each iteration of the REPEAT statement, the key variable is incremented and tested to see if all 20 function keys have been stored. If not, it's used in the next iteration to indicate which function key to store. As soon as the 20th function key is stored, the key variable is incremented to 20, and the REPEAT statement ends. After that, we close the disk file and return a value of 0 to whatever function called save_fkeys() to indicate success.

```
//////////////////////////////////////////////
//   restore_fkeys();
//
//   Syntax: status = restore_fkeys(STRING file_name);
//
//   Restores the STalker function key strings from data stored in the
// file. The file must be in the format "key_number,
// key_string_length, key_string", as stored by the save_fkeys()
// function.
//////////////////////////////////////////////
FUNCTION restore_fkeys(string file_name) RETURNS INT
INT handle;
INT index;
INT key_size;
INT key;
STRING key_buffer[41];
IF not file_exists(file_name) THEN
   return -1;          // file doesn't exist
ENDIF
IF (handle=file_open(file_name, FO_READ)) < 0 THEN
   return handle;          // file open error
ENDIF
// read the keys from the file in order
key=0;
REPEAT
   file_getchar(handle);   // function key number
   key_size=file_getchar(handle);
   file_read(handle, key_buffer, key_size);
   key_buffer[key_size]=0;
   set_string_param(FKEY_PARAM, key, key_buffer);
```

```
   key=key+1;
UNTIL key == 20;
// close up, return success
file_close(handle);
return 0;
ENDFUNCTION
```

The `restore_fkeys()` function begins by declaring similar variables, including a `key_size` integer to hold the size of the function key string being read from disk. It checks for the existence of the indicated file, then tries to open it, and returns an error if either operation fails.

Then it uses a similar REPEAT statement to read each function key string from the file and insert it into its appropriate function key. First, the key number is read using a `file_getchar()` call that does nothing with the value. The key number is useless right now, but we still must account for it in the file. The size of the string is then read into the `key_size` variable, and `file_read()` is used to read in the number of characters indicated by `key_size` and store them in the `key_buffer` string.

The next step is important. A value of zero is stored just after the last character of the function key string in key_buffer to null-terminate it. This is required so that the `set_string_param()` function can tell how long the function key string is, in case it is shorter than 40 characters, which is most of the time. (Since the first character is `key_buffer[0]`, `key_size` points to the array position just after the last character of the function key string.)

After the `set_string_param()` function inserts the string into the function key, the `key` variable is incremented and tested to see if all the function keys have been read. When `key` is 20, the REPEAT statement ends, the file is closed, and the function returns a value of 0 indicating success.

The `key_buffer` string is an example of when you should use a string instead of a string pointer. In this case, we are reserving space in memory to be used by the `get_string_param()` and `set_string_param()` functions, instead of using it as a quasi-constant.

Now, the logon script looks like this:

```
FUNCTION save_fkeys RETURNS int;
FUNCTION main()
STRING   log_id;
STRING   pword;
STRING   tempkey;
STRING   bye_string;
   log_id     = "myname\r";
   pword      = "myword\r";
   tempkey    = "c:\\tempkey.stk"
   bye_string = "\xDDbixoff.btk"
   IF not port_connected() THEN
```

```
            puts("\r\nDialing Bix+\r\n");
            IF dial_one("bix") THEN
               exit -1;
            ENDIF
         ENDIF
         REPEAT
            port_putchar('a');
         UNTIL wait_for("log in:", 5);
         REPEAT
            port_puts("bix\r");
         UNTIL wait_for("call connected", 50);
         wait_for("Name?", 0);
         port_puts(log_id);
         wait_for("word:", 0);
         port_puts(pword);
         IF save_fkeys(tempkey) == 0 THEN
            set_string_param(FKEY_PARAM, 9, bye_string);
         ENDIF
      ENDFUNCTION
      // Include the function definition for save_fkeys() here!
```

At the very top, we've added a function declaration for our `save_fkeys()` function, and indicated that the definition should be added at the bottom. Remember that functions must be either declared or defined before the are called in the script. It is preferable to have declarations at the top, followed by the `main()` function, followed by other functions. We could have just as easily placed the function definition for the `save_fkeys()` function above the `main()` function and done without the function declaration.

Another string pointer, `tempkey`, has been added and assigned to point to the filename we want to use to store the function keys. (Note the double backslash in the path.) `Bye_string` was changed so that when it is executed as a function key it calls the script named "BIXOFF.BTK" (your logon script might then be called "BIXON.BTK"). Then we added an IF statement at the bottom of the function to check the return value from `save_fkeys()`. If it returns 0, the function keys have been successfully stored and it's safe to change F10. Otherwise, no change is made.

Now that you have walked through a Bixon script, you should be able to create a Bixoff script. A sample has been included at the end of this chapter for reference.

3.3. Reading a String of Arbitrary Length

Anyone who has actually used BIX for very long will tell you that logging on is not always a straightforward process. If BIX is busy, it may not allow you to log on and will return you to the Tymnet "`please log in:`" prompt. In that case, the REPEAT statement that waits for "`call connected`" will continue to blindly send "`bix\r`"

every five seconds whether Tymnet is listening or not. Sometimes the BIX computer answers the call, but then informs you that the system is inoperative and again returns you to the Tymnet prompt. In that case, our script will wait forever for the "Name?" prompt that never appears, while Tymnet is waiting for a login name. (Actually, Tymnet will hang up after a while if it doesn't get the login name, but our script will still be waiting diligently.)

It would be nice if instead of waiting for a specific prompt, the script could read the prompts it does get and respond accordingly. Unfortunately, prompts rarely end in carriage returns or line feeds. Since `port_gets()` only works with lines ending in a line feed and `port_read()` only gets a fixed number of characters, we can't use either to store the prompt to a string so the script can examine it. In fact, there is no built-in function to obtain a character string of arbitrary length without a line ending from the communications port. We'll have to write our own.

```
////////////////////////////////////////////
//  port_read_string();
//
//  Syntax: buffer = port_read_string(STRING buffer, INT buf_size
//                                    INT timeout);
//
//  Reads a line or partial line from the comm port and returns
//  it in buffer. Differs from port_gets() in that it will
//  return when a newline is received or after waiting 'timeout'
//  seconds with no input.
////////////////////////////////////////////
FUNCTION port_read_string(
   STRING   buffer,    //storage buffer, will be NULL terminated
   INT      buf_size,  //buffer size
   INT      timeout)   //timeout value in seconds
   RETURNS   STRING    //return buffer address for convenience
INT   timer;
INT   index;
INT   done;
// Need room for at least a \n and null in the line buffer
IF buf_size < 2 THEN
    return NULL;            // bad buffer
ENDIF
index=-1;
done=FALSE;
timer = timer_set(timeout*10);   // set a timer
// fill the buffer until \n is received or timeout
WHILE not timer_check(timer) and not done and (index < buf_size) DO
    WHILE port_instat() and not done and (index < buf_size) DO
        done=(buffer[index=index+1]=port_getchar())=='\n';
    ENDWHILE
ENDWHILE
buffer[index+1]=0;   // terminate the buffer
return buffer;       // returns the same pointer it was passed
ENDFUNCTION
```

The function `port_read_string()` takes a pointer to a buffer to put the received line into, the length of the buffer, and a length of time in seconds to wait for an end of line before returning. It fills the buffer with characters from the communications

port until a line feed character is received, the buffer fills up, or the host stops sending for 'timeout' seconds.

The buffer size is passed into the function because otherwise the function cannot tell how big the buffer is. The parameter passed is just a string pointer; sizeof() will always return 4 for a pointer, and the return value of strlen() depends on the contents of the buffer, not the buffer itself. The function has to insure that it doesn't overfill the buffer because the excess will corrupt memory somewhere and cause strange and mysterious malfunctions.

The buffer size is checked to insure it has room for at least one character, then the control variables index and done are initialized, and the timer is set with the timeout parameter. The variable done will be used as a flag, either TRUE or FALSE.

Next are two nested WHILE statements. Both statements check the done flag (explained later) and whether the buffer is full (index points to the last character in the buffer). The outside WHILE checks the timer, and the inside WHILE checks the communications port for ready characters.

The meat of the function is the single line within the inner WHILE statement. To make it easier to read, let's expand it into three lines:

```
index = index +1
buffer[index] = port_getchar()
done = (buffer[index] == '\n')
```

First, index is incremented to point to the next character position in the buffer. (That's why it was initialized to -1, so that the first time through it would point to character 0). The next character in the communications port is fetched and put into position in the buffer using port_getchar(). Since port_getchar() could wait indefinitely if there wasn't already a character waiting at the communications port, we test port_instat() in the WHILE statement. Then we check to see if the character was a line feed ("\n", ASCII 10) and set done equal to the result. If it was a line feed, done will be true.

The reason we joined those three statements into one compound statement is to squeeze just a little bit more speed out of the function. Since the BackTALK compiler is a non-optimizing compiler, these two ways of accomplishing the same thing actually produce slightly different code in the compiled script. The improvement in speed is slight, but these operations are carried out every time a character is received from the communications port. A small improvement for each character results in significant improvement overall.

Then the inner WHILE statement checks the done flag, index, and the communications port again. If the done flag is FALSE (meaning a line feed has not yet been received), if index is still smaller than the buffer size (meaning the buffer is not full), and

if there is another character waiting in the communications port, the process is repeated and the buffer fills with characters.

When a line feed is received, the buffer gets full, or if all the characters in the communications buffer are processed, the inner WHILE statement will pass control back to the outer WHILE. If control was passed because `done` was TRUE or the buffer was full, that WHILE exits and the rest of the script is executed. If not, and if the timer is not expired, the inner WHILE loop gets control again and looks for more characters. This process repeats until the timer expires.

The last thing the function does before returning is to null-terminate the buffer. Since `index` points to the last character in the buffer, we set the character at `index+1` to 0 so that any other function that uses the buffer can tell how long the string is.

Note that the function returns the same string pointer that it was passed, something that the calling function surely knows already. This is a common practice, and makes using the function much more convenient. For example, it allows statements like:

```
puts(port_read_string(buffer, sizeof(buffer), 2));
```

In which the pointer that `port_read_string()` returns becomes the argument to `puts()`. If `port_read_string()` did not return a pointer, the code would have to look like:

```
port_read_string(buffer, sizeof(buffer), 2);
puts(buffer);
```

Now the Bixon script can look like this:

```
FUNCTION save_fkeys RETURNS int;
FUNCTION port_read_string RETURNS string;
FUNCTION main()
INT   logged_on;
STRING   log_id;
STRING   pword;
STRING   tempkey;
STRING   bye_string;
STRING   in_line[80];
   log_id     = "myname\r";
   pword      = "myword\r";
   tempkey    = "c:\\tempkey.stk"
   bye_string = "\xDDbixoff.btk"
   IF not port_connected() THEN
      puts("\r\nDialing Bix+\r\n");
      IF dial_one("bix") THEN
         exit -1;
      ENDIF
   ENDIF
   REPEAT
      port_putchar('a');
   UNTIL wait_for("log in:", 5);
   logged_on = FALSE;
   REPEAT   // Until logged_on == TRUE
   port_read_string(in_line,sizeof(in_line),5);   // wait for next line
   puts(in_line);         // echo the line to the terminal window
```

```
      // TYMNET+ log in prompt
      IF strstr(in_line, "log in:")!=NULL THEN
         port_puts("bix\r");
         port_gets(in_line, sizeof(in_line));   // trap the echo
         puts(in_line);                         // print the echo
      ENDIF
      // BIX+ login prompt
      IF strstr(in_line, "Name?")!=NULL THEN
         port_puts(log_id);
         port_gets(in_line, sizeof(in_line));   // trap the echo
         puts(in_line);                         // print the echo
      ENDIF
      // BIX+ password prompt
      IF strstr(in_line, "Password:")!=NULL THEN
         port_puts(pword);
         port_gets(in_line, sizeof(in_line));   // trap the echo
         puts(in_line);                         // print the echo
         logged_on = TRUE;                      // all done
      ENDIF
      // if the host hung up, exit
      IF not port_carrier() THEN
         exit -1;
      ENDIF
      UNTIL logged_on;
      IF save_fkeys(tempkey) == 0 THEN
         set_string_param(FKEY_PARAM, 9, bye_string);
      ENDIF
   ENDFUNCTION
   // Include the function definition for save_fkeys() here!
   // Include the function definition for port_read_string() here!
```

A function declaration for `port_read_string()` has been added, as well as two more variables: `logged_on` which is used as a TRUE/FALSE flag, and `in_line` which is used as the buffer for `port_read_string()`. We set `logged_on` to FALSE and replaced the REPEAT statement that waits for the "call connected" string with another REPEAT statement that waits for `logged_on` to be TRUE. Inside that REPEAT, we use `port_read_string()` to put every line that arrives into the `in_line` string. Then we check `in_line` for each of the prompts we need to respond to using IF statements and `strstr()` and respond appropriately. Finally, after the password prompt has been received and the password sent, `logged_on` is set to TRUE and the REPEAT statement ends. The function keys are saved and `F10` is set as before.

Note that we check `port_carrier()` inside the REPEAT statement. If BIX steadily refuses access and our script persists long enough in trying to log on, Tymnet will hang up on us. In that case, the script simply exits.

Also note that we are now using `puts()` to print `in_line` to the terminal window. Since we are no longer using `wait_for()`, the script must take care of all processing of the incoming data. It even has to collect the echo of the commands sent back by BIX and print it to the window. We can use `port_gets()` for that, since we know that the echoes will all end in a linefeed. If you access BIX in "half-duplex" or "local" mode, BIX doesn't send echoes and you should delete that part.

3.3.1. More capabilities

Now that we have a good basic logon script, let's add a few bells and whistles. There may be occasions when you'd like to have the script log on and do something online all by itself, maybe download files in the middle of the night. It might be nice to have a general purpose function that would interact with the user to find out what time the event should happen and then wait until then.

```
////////////////////////////////////////////
// get_wait()
//
// Syntax: success = get_wait(STRING prompt);
//
// Interacts with the user to get a date/time or a fixed delay
// interval and waits for completion. The prompt string is displayed
// in the alert box presented, and should be formatted with vertical
// bars indicating line breaks. Prompt should not exceed 30
// characters between line breaks or 5 lines. Returns TRUE if the
// wait was completed (including when the user indicates no wait)
// and returns FALSE if the wait was aborted with the UNDO key.
////////////////////////////////////////////
FUNCTION get_wait(STRING prompt) RETURNS INT
INT   button;
INT   valid;
INT   hour;
INT   minute;
STRING   al_string[180];
STRING   response[6];
STRING   scrap[3];

// build the alert string and call the alert box
sprintf(al_string, "[3][%s][Stop|Go|Cancel]", prompt);
button = alert(al_string);
// if the user chose "Go", then return TRUE without delay
IF button == 2 THEN
   return TRUE;
ENDIF
// if the user chose "Cancel", then return FALSE
IF button == 3 THEN
   return FALSE;
ENDIF
// Print a prompt and wait for a keypress
puts("\r\nWait for clock [T]ime, Fixed [D]elay, or [Q]uit? ");
REPEAT
   putchar('\a');
   button = getchar();
   IF islower(button) THEN
      button = button - 32;   // convert lower case characters
   ENDIF                      // to upper case
UNTIL button == 'T' or button == 'D' or button == 'Q';
putchar(button);      // echo the response to the terminal window
// if the user chose "Quit", return FALSE
IF button == 'Q' THEN
   return FALSE;
ENDIF
// get a valid HH:MM input
REPEAT
   IF button == 'T' THEN
      puts(\r\nEnter the time of day in 24 hour format\r\n\
```

```
as HH:MM (00:00 to Cancel) ");
   ENDIF
   IF button == 'D' THEN
      puts("\r\nEnter the delay in hours and minutes\r\n\
as HH:MM (00:00 to Cancel)");
   ENDIF
   gets(response, sizeof(response));
   IF response[0] == '0' and
      response[1] == '0' and
      response[3] == '0' and
      response[4] == '0' THEN
      return FALSE;   // cancel when all zeroes are entered
   ENDIF
   valid = isdigit(response[0]) and
         isdigit(response[1]) and
         response[2] == ':'   and
         isdigit(response[3]) and
         isdigit(response[4]);      // correct format, valid = TRUE
UNTIL valid;
// calculate hour and minute integers from string input
scrap[0] = response[0];
scrap[1] = response[1];
scrap[2] = 0;
hour = atoi(scrap);
scrap[0] = response[3];
scrap[1] = response[4];
minute = atoi(scrap);
// do the appropriate wait
IF button == 1 THEN
   return wait_until(-1, -1, -1, hour, minute);
ENDIF
IF button == 2 THEN
   return wait(10*(360*hour+60*minute), TRUE);
ENDIF
ENDFUNCTION
```

We start by declaring variables. The integer `button` will hold the number of the button selected by the user in an alert and `valid` will be used as a TRUE/FALSE flag. The string `al_string` will hold the argument passed to the `alert()` function, and `response` will hold the user input for delay period or time of day. The `hour` and `minute` integers will be calculated from the user response and given to the `wait_()` functions, and the `scrap` string is used in the calculation.

Then we use the `sprintf()` function to construct a formatted string containing the elements required by `alert()`. We're assuming that the prompt string sent by whatever function calls `get_wait()` is correctly formatted. Then `alert()` is called to give the user the choice between continuing immediately or delaying. The number of the button chosen in the alert is returned and placed in `button`, and tested. If the user selected "Go" (button 2), the function returns TRUE to tell the calling function to continue with whatever it was doing. If the user selected 'Cancel' (button 3), the function returns a FALSE, to tell the calling function to abort.

If the user elected to wait, we print a prompt in the terminal window using `puts()` asking the user for a single keypress. Inside a REPEAT statement, we print a "\a" character to the terminal window using `putchar()` to ring the bell and alert the user.

Next, the `getchar()` function waits for the user to press a key and then stores the ASCII value in button, converting it to upper case in the process. Then the REPEAT statement tests `button` to see if it is one of the acceptable responses. If it is, the function continues. If not, the REPEAT statement loops, rings the bell, and waits for user input again.

Then a REPEAT statement starts that repeats until `valid` becomes TRUE. Inside the REPEAT statement, we display a prompt that depends on the user's selection in the alert. The `gets()` function is used to collect the user's input for either time of day or delay period. After we get the input, we have to check it to make sure it's understandable. First, if all the digits are zeroes, the function returns FALSE. This gives another way to cancel the function. Second, we make sure that the first, second, fourth and fifth characters are numeric and that the third character is a colon (`:`). If all that is true, then we set the valid flag to TRUE, and the REPEAT statement ends. Otherwise `valid` is FALSE and the REPEAT statement loops, asking for input until it gets either valid input or the indication to cancel (all zeroes).

Once valid input has been given, we have to convert the input string to integers to be used in the `wait_()` functions. The characters from `response` are put into the `scrap` string two at a time and the `atoi()` function returns the `hour` and `minute` integers.

Finally, depending on the value of `button` either a `wait()` or a `wait_until()` function is called using the user's input. The `get_wait()` function returns the value returned from the `wait_()` function. If the user presses [Undo] during the wait the function returns FALSE and successful waits return TRUE. The `wait()` function is called with its second parameter TRUE so that incoming characters will be processed by STalker normally during the wait.

The `get_wait()` function illustrates some ways of handling user interaction in BackTALK scripts. The GEM alert is a convenient way of offering up to three choices, but bogs down when the choices are more complex and doesn't work at all for string inputs. More choices can be offered if you use `getchar()` to collect a single keypress. In that case, though, you need to provide the validation yourself to insure responses other than the ones you are prepared for are ignored. In this case, we also ring the terminal bell to tell the user that the input was not accepted. Likewise, the prompt should make clear what characters are accepted as input, as we did in the previous script with square brackets, "`[]`", around capital letters. You also have to decide how to handle upper and lower case in your responses. Here, we just converted all lower case responses to upper case. Finally, the `gets()` function accepts a character string and even allows the user to backspace and delete to correct the input. Use this when you need specific information, such as a time. In general, use `getchar()` when you have a list of menu items to choose from (and keep your menus as short as possible), and use `gets()` when you need specific input. Using `gets()` for single character input is inconvenient for the user since they have to press [Return] to complete the entry and inconvenient for you since it is more work to validate the input.

Now, after all that, we can use `get_wait()` in our Bixon script:

```
IF not port_connected() THEN
   IF not get_wait("Dialing Bix+") THEN
      exit 1;
   ENDIF
   puts("\r\nDialing Bix+\r\n");
   IF dial_one("bix") THEN
      exit 1;
   ENDIF
ENDIF
```

This is just the fragment of our bixon script that checks the `port_connected()` status. It doesn't make any sense to display the prompt and wait unless the script called on its own, so the `get_wait()` function is called only if `port_connected()` is FALSE. The return from the `get_wait()` function is tested in another IF statement, and if it is FALSE the script exits. If `get_wait()` returns TRUE (the user chose not to wait, or the wait was successful), the script dials BIX and proceeds normally.

Remember to declare the `get_wait()` function at the top of the bixon script with a statement like:

```
FUNCTION get_wait RETURNS int;
```

and add the function declaration to the bottom.

3.4. More Example Scripts

3.4.1. BIXOFF.BTS

```
// bixoff.bts
////////////////////////////////////////////
//   Logs off BIX+ by sending the 'bye' command, waiting for the
//   Tymnet+ 'please login' prompt and sending ^Z (26) to hang up
//   Tymnet+. Restores function keys from G:\TEMP.KEY if they were
//   saved.
////////////////////////////////////////////
INCLUDE "stalker.bth"
// declare the restore_fkeys() function, defined after main()
FUNCTION restore_fkeys RETURNS int;
FUNCTION main()
    // send 'bye', wait for the Tymnet+ prompt, and send ^Z to hang up
    // Tymnet+
    port_puts("bye\r");
    wait_for("log in:", 0);
    port_putchar(26);
    // try to restore the function keys. if not successful, print a
    // message on the terminal screen
    IF restore_fkeys("g:\\temp.key") < 0 THEN
        puts("\r\nFunction keys not restored.\r\n");
    ENDIF
ENDFUNCTION
```

```
////////////////////////////////////////////
//   restore_fkeys();
//
//   Syntax: status = restore_fkeys(STRING file_name);
//
//   Restores the STalker function key strings from data stored in
// the file. The file must be in the format "key_string_length,
// key_string", as stored by the save_fkeys() function.
////////////////////////////////////////////
FUNCTION restore_fkeys(string file_name) RETURNS INT
INT handle;
INT index;
INT key_size;
INT key;
STRING key_buffer[41];
IF not file_exists(file_name) THEN
   return -1;          // file doesn't exist
ENDIF
IF handle=file_open(file_name, FO_READ) < 0 THEN
   return handle;          // file open error
ENDIF
// read the keys from the file in order
key=0;
REPEAT
   file_getchar(handle);   // function key number
   key_size=file_getchar(handle);
   file_read(handle, key_buffer, key_size);
   key_buffer[key_size]=0;
   set_string_param(FKEY_PARAM, key, key_buffer);
   key=key+1;
UNTIL key == 20;
// close up, return success
file_close(handle);
return 0;
ENDFUNCTION
// end bixoff.bts
```

3.4.2. FKEYS.BTS

```
// fkeys.bts
////////////////////////////////////////////
//   Saves or restores function key settings.
////////////////////////////////////////////
INCLUDE "stalker.bth"
////////////////////////////////////////////
//  save_fkeys();
//
//  Syntax: status = save_fkeys(STRING file_name);
//
//  Saves the contents of the STalker function key strings in the
//  file. The file is in the format "key_string_length, key_string".
//  Returns 0 if successful, or the negative GEMDOS error code.
////////////////////////////////////////////
function save_fkeys(string file_name) returns INT
INT handle;
INT key;
STRING key_buffer[41];
key=0;
if handle=file_create(file_name) < 0 then
   return handle;
```

```
endif
repeat
   get_string_param(FKEY_PARAM, key, key_buffer, sizeof(key_buffer));
   file_putchar(handle, strlen(key_buffer));
   file_puts(key_buffer, handle);
   key=key+1;
until key == 20;
file_close(handle);
return 0;
endfunction

//////////////////////////////////////////////
//   restore_fkeys();
//
//   Syntax: status = restore_fkeys(STRING file_name);
//
//   Restores the STalker function key strings from data stored in the
// file. The file must be in the format "key_number,
// key_string_length, key_string", as stored by the save_fkeys()
// function.
//////////////////////////////////////////////
FUNCTION restore_fkeys(string file_name) RETURNS INT
INT handle;
INT index;
INT key_size;
INT key;
STRING key_buffer[41];
IF not file_exists(file_name) THEN
   return -1;          // file doesn't exist
ENDIF
IF handle=file_open(file_name, FO_READ) < 0 THEN
   return handle;          // file open error
ENDIF
// read the keys from the file in order
key=0;
REPEAT
   file_getchar(handle);   // function key number
   key_size=file_getchar(handle);
   file_read(handle, key_buffer, key_size);
   key_buffer[key_size]=0;
   set_string_param(FKEY_PARAM, key, key_buffer);
   key=key+1;
UNTIL key == 20;
// close up, return success
file_close(handle);
return 0;
ENDFUNCTION
function main()
INT   response;
INT     fsel_button;
INT   alert_button;
STRING  filename[14];
STRING  keyfile[128];
STRING  pathname[128];
STRING  promptstring;
alert_button = alert("[0][Function Key Loader][Quit|Save|Restore]");
if alert_button == 1 then
   return;
endif
if alert_button == 3 then
   promptstring = "Restoring Function Keys:";
```

```
else
   promptstring = "Saving Function Keys:";
endif
pathname[0] = file_get_drive();
pathname[1] = ':';
strcat(file_get_dir(0, pathname + 2), "\\*.KEY");
if !fsel(pathname, filename, keyfile, promptstring) then
   return;
endif
if alert_button == 3 then
   if restore_fkeys(keyfile) < 0 then
      alert("[3][ | Function keys not restored! ][Ok]");
   else
      alert("[1][ |   Restored!  ][Ok]");
   endif
else
   if save_fkeys(keyfile) < 0 then
      alert("[3][ |   Function keys not saved!  ][Ok]");
   else
      alert("[1][ |    Saved!    ][Ok]");
   endif
endif
endfunction
// end fkeys.bts
```

4. BackTALK Reference Manual

This reference manual covers all elements of the BackTALK language. If you are an experienced programmer, you should be able to find any information you need in this section without having to refer to the tutorial chapters.

BackTALK is very similar to the C programming language, and programmers familiar with C may be able to use their knowledge to implement BackTALK more effectively.

4.1. Scripts

BackTALK is a way of writing scripts for use with STalker version 3.0 and above. Scripts are between macros, which simulate keyboard and mouse actions and let the program do the work, and programs which control what the computer does at the most basic level. A BackTALK script tells STalker what to do and when to do it and allows STalker to make some basic decisions about what action to take.

BackTALK is a compiled script language, which means that scripts must be processed by the BackTALK compiler before they can be used by STalker. See Appendix E for details on operating the compiler.

There are seven ways to run a BackTALK script in STalker:

- Pressing [Alternate][Enter] when the STalker window is active or selecting *Run BackTALK Script* from the **BackTALK** pull-down menu will bring up a file selector so the user can choose a BackTALK script to execute.
- Up to ten scripts may be listed under the **BackTALK** pull-down menu, and may then be run by selecting their menu entries or by pressing the corresponding number on the numeric keypad while holding down the [Alternate] or [Control] keys.
- Entering the file name of the BackTALK script in a STalker function key string preceded by a [Shift][Clr/Home] character tells STalker to run the script as part of the function key macro.
- Each entry in the autodialer can be assigned a script by clicking on the "S" box. The script file must be stored in the default script directory, set in the STalker Preference Settings dialog.

- During computer startup when STalker is initially loaded, STalker checks the default script directory for the script "STK_STRT.BTK" and runs it if present.
- Similarly, the first time the window is opened STalker will execute the script "STK_OPEN.BTK" if it is present. It is fine to have both present.
- When STalker's window is closed, it will run a script called "STK_EXIT.BTK", if present.

4.2. Terms

The examples here include some syntax notation to describe what must go into the statements. Items enclosed in square brackets, e.g. [parameter-list], are not always required. Items enclosed in angle brackets, e.g. <variable-name> are always required. When writing scripts, the programmer supplies the names or values, not including the square or angle brackets.

An *identifier* is the name given by the programmer to a variable, constant, or function in the script. A keyword is the name of a statement, built-in function or constant. Identifiers can be up to 32 characters long, but BackTALK only looks at the first 16. Identifiers must start with a non-digit character, and may contain alphabetic and numeric characters and the underscore ("_"). Upper and lower case make a difference, so "zowie", "Zowie" and "ZOWIE" are all different. BackTALK pays no attention to upper and lower case in keywords or the named of pre-defined functions.

An *expression* is any combination of constants, variables, and function calls linked together that result in a value. *Statements* are commands to accomplish something and are made up of one or more expressions.

BackTALK source files are simple text files containing BackTALK statements. Any word processor that can save text as plain ASCII can be used to write script source files, but STeno is ideal. Tabs, spaces and carriage returns (line endings) are ignored by the BackTALK compiler, so they can be used to format the file in whatever manner is most readable. All other characters are significant and play a part in the BackTALK language. For example:

```
IF first < second THEN puts("Wow"); ENDIF
```

is equivalent to

```
IF first < second THEN
   puts("Wow");
ENDIF
```

and

```
IF first <
   second          THEN
```

```
    puts  (      "Wow" )     ;
ENDIF
```

Appendix C includes some hints on formatting source code that are more or less standard and make the text as readable as possible. Since carriage returns are not significant to the compiler, all statements must end with a semicolon (";"). This is one of the most common sources of trouble for persons unfamiliar with BackTALK. Exceptions are the logic structures (IF, THEN, ENDIF, etc.) and comments. Both will be discussed later in this manual.

Those already familiar with the C language will find BackTALK very familiar. The BackTALK language is intentionally similar to C to take advantage of its power and flexibility. There are differences and limitations, however, that you should be aware of, so read the reference section carefully.

4.3. Data

BackTALK supports two kinds of data: integers and strings. Integers are 4 byte, 32 bit numbers, also known as long integers in other languages. Strings are arrays of bytes referred to by a single identifier. All variables and constants that are declared inside functions are local to that function and can not be accessed by any other function. Local variables cease to exist once the function returns, and any values stored in them are lost. Variables or constants declared outside any function are global to the entire script, and are available for use and modification by any function.

Variable and constant names must be declared before they are used in the script. The declaration tells the BackTALK compiler to set aside room for the variable in memory, and attaches the variable name to that memory.

4.3.1. Integers

Integers are declared using the INT keyword followed by the variable name. Only one variable can be declared on a line. For example:

```
INT number;
```

Declaring a variable as an integer allows it to be a number between -2,147,483,648 and 2,147,483,647 (the 32nd bit is used as the sign flag). Any attempt to use a larger number results in 'lost' bits and incorrect results.

Integer variables can also be used to hold ASCII characters, since ASCII characters are really just integers in the range 0-255. The statements

```
variable_one = 65;
```

and

```
variable_one = 'A';
```

mean exactly the same thing to BackTALK. Any ASCII character enclosed in single quotes is equivalent to its ASCII value. (A table of ASCII values for each character is included in Appendix A.) The single quotes are important, since double quotes indicate a string, and will be interpreted differently.

Characters may be specified using an ASCII character in single quotes as above, or using special character sequences allowing the use of non-printable characters:

Sequence	Name	ASCII Character	Value
\a	Audible Bell	[Control][G]	7
\b	Backspace	[Control][H]	8
\f	Form Feed	[Control][L]	12
\n	NewLine (LF)	[Control][J]	10
\r	Carriage Return	[Control][M]	13
\t	Tab	[Control][I]	9
\v	Vertical Tab	[Control][K]	11
\'	Single Quote	'	39
\\	Backslash	\	92
\DDD	1 to 3 Octal Digits	DDD	
\xDD	1 to 2 Hex Digits	xDD	

Whenever the BackTALK compiler is expecting a character and sees one of these combinations, it substitutes the ASCII value. For example, "\r" represents the carriage return character, and the statement

```
variable_one = '\r';
```

would set the value of "variable_one" to 13. The "\\" pair is used to denote a single backslash, e.g. in pathnames. Any character may be entered by using the backslash and three octal digits or \x and two hexadecimal digits. Be careful when using these in strings. For example,

```
"\xFFor once and for all."
```

will result in

```
or once and for all.
```

with an ASCII 0xFF at the front since BackTALK takes the first two valid hexadecimal digits it finds. You can avoid the problem by using a leading zero, i.e.

```
"\x0FFor once and for all."
```

4.3.2. Constants

Numeric constants can be entered in decimal, hexadecimal, or octal notation, depending on the leading digits. For example,

```
age = 47;
age = 0x2F;
```

```
age = 057;
```

all assign the value 47 to age. Hexadecimal numbers are preceded by 0x or 0X, and octal numbers are preceded by 0.

There are four constant names built into BackTALK: TRUE and ON are defined as the number 1, and FALSE and OFF are defined as the number 0.

A CONSTANT is used to give more readable names to numeric values that are used in the script. It may not change its contents once it is declared, and may only contain an integer. The declaration for a constant must always begin with CONSTANT INT followed by the constant name and the value to be assigned to it. For example:

```
CONSTANT INT constant_one = 12;
```

When the script is compiled, the numeric value is substituted for the constant name wherever it appears -- constant definitions take up no extra space in the compiled script.

4.3.3. Strings

Strings are declared using the STRING keyword followed by the variable name and the size of the string inclosed in brackets. For example:

```
STRING word[21];
```

declares a string called word and allows it to be 21 bytes, or characters, long.

A STRING is a string of characters (a one-dimensional array). BackTALK marks the end of a string with an ASCII 0 (null) as does C. Therefore, the size of a string should always be one greater than the maximum number of characters it will contain. Strings can be as large as needed, limited by the size of the STalker script execution buffer (64k maximum). Individual characters in a string can be accessed using conventional array notation. For example, if name contains "Mark Twain", name[5] indicates a capital 'T'.

Each element of the string is a byte, 8 bits, instead of the 32 bits of an integer. If the element is referenced alone, like

```
x = string[1];
```

it is automatically converted (or cast) into a 32-bit integer with zeroes in the upper 24 bits. If an integer is assigned to a string element, like

```
string[1] = x;
```

only the lower 8 bits are assigned.

When read and write functions are used with strings that may contain "binary" data, or data that doesn't necessarily correspond to ASCII characters, be sure to use

functions intended for binary data. Otherwise, the function may be confused by any '0' bytes embedded in the string.

There is one string constant defined, "NULL". This is a special type of string that is interpreted as zero by some functions.

4.3.4. String pointers

The statement

```
STRING variable_three;
```

has no array size in square brackets, which means that it defines a special kind of number called a string pointer. String pointers point to a string; an indirect reference. String pointers are the way that one function in BackTALK tells another function to use the data in a string without making a copy of it. Once the pointer is declared, it can be assigned a value from another string or pointer.

4.3.5. A note about strings

(Experienced C programmers can probably skip this section.)

Strings are very powerful and yet very troublesome to novices. It's helpful to remember a few things about strings:

Because strings can be arbitrarily large, it is impractical for BackTALK to make a copy of a string to pass back and forth between functions. Therefore, functions only pass string pointers back and forth. In order for functions to return string information, such as a file name or user input, the string must be declared outside the function and the pointer given to the function as a parameter. The function can modify the string and then return the pointer. Strings declared inside functions are lost once that function returns.

String literals, e.g. "characters between double quotes", can be used wherever the language expects a string. The compiler stores the string in memory and gives the pointer to the function that needs it. String literals can only be up to 512 bytes long.

To split a string across one or more lines in a script file, use the backslash character "\" as the last character on the line which is being split.

```
   printf("This string is \
being split because it is too long.");
```

Values can not be assigned to strings with the "=" operator. In order to place characters in a string, use the `strcpy()` function. An alternative is to declare the name of the string as a pointer, then assign the pointer to a string literal. For example:

```
STRING word;
word = "supercalafragalisticexpialadotious";
```

has a similar effect to

```
STRING word[35];
strcpy(word, "supercalafragalisticexpialadotious");
```

When using this technique remember to make the initial string literal as long as it will ever need to be.

4.4. Operators

Operators are instructions to BackTALK on what to do to the various constants, variables, and function calls to result in a value. The following operators go between two expressions (binary operators):

Operator	Name	Category
+	addition	arithmetic
-	subtraction	arithmetic
*	multiplication	arithmetic
/	division	arithmetic
%	remainder	arithmetic
& or BITAND	bit-wise AND	arithmetic
| or BITOR	bit-wise OR	arithmetic
^ or BITXOR	bit-wise exclusive OR	arithmetic
>>	bit-shift right	arithmetic
<<	bit-shift left	arithmetic
&& or AND	logical AND	logical
|| or OR	logical OR	logical
>	greater than	logical
>=	greater than or equal to	logical
<	less than	logical
<=	less than or equal to	logical
==	equal to	logical
!= or <>	not equal to	logical
=	assignment	

and these operators go in front of an expression (unary operators):

Operator	Name	Category
-	negative	arithmetic
~ or BITNOT	bit-wise NOT	arithmetic
! or NOT	logical NOT	logical

Two names are given to some of the operators for convenience to C programmers. The first name is the familiar C operator and the second is somewhat more descriptive of what the operator actually does. Either may be used in scripts.

Addition, subtraction, multiplication, and division are conventional arithmetic operators that take two integers and produce a third. Since BackTALK doesn't support

fractions, division always results in the integer portion of the quotient. The logical operators treat the expressions on either side as either TRUE (non-zero) or FALSE (zero), and produce a result of either TRUE or ON (one) or FALSE or OFF (zero). The logical operators work only on integers; to compare strings use the string handling functions. The NOT operator inverts the logical sense of its operand, turning FALSE to TRUE and vice versa.

Note that the logical test of equality, "==", is different than the assignment operator, "=". This is unlike languages such as BASIC in which the syntax requirements are less strict.

The remainder operator, %, is also known as the modulus. It divides the first number by the second as in division, but results in the remainder of the division. For example, the expression `7 % 2` results in `1`, the remainder after 7 is divided by 2. If the remainder of a quotient is zero, numbers are evenly divisible.

The "bit-wise" operators treat each bit of an integer separately. These operators are part of what give the BackTALK language its economy of memory and speed of execution, since complex logical operations can be done using individual bits instead of complete integers.

The BITAND operator performs a logical AND between each bit of two integers. This is primarily useful for "masking" numbers. By BITANDing an arbitrary integer with one of a known pattern, only the bit positions that are set in both numbers will be set in the result.

The BITOR operator performs a logical OR between each bit of the two integers. This is handy for "stacking" bit patterns on top of each other since the resulting bit position will be set if the bits in either of the operands was set.

The BITXOR operator performs an exclusive OR between each bit of the two integers. Exclusive OR results in a bit set if either one of the operand bits are set, but not both. This has the effect of laying the patterns together and 'blanking out' the bits where they match.

The "bit-shift" operators "move" the bit pattern of the first operand right or left by the number of positions specified by the second operand. This is most useful in extracting fields of information stored in a single integer, such as the return from the `get_time()` function. Bit positions vacated at the end of the bit pattern are filled with "0"s. See the description of the `get_time()` function for an example.

The BITNOT operator simply inverts each bit of the operand.

The operators are evaluated according to their precedence, or the priority that the BackTALK compiler places on them. The precedence is, from highest to lowest:

```
-   !   ~                          (unary operators)
*   /   %
+   -
<<, >>
>   >=  <   <=  ==  !=
&
^
|
&&
||
=                                  (assignment)
```

Parts of an expression in parentheses, "()", are always evaluated first. For example,

```
a + b * c
```

would be evaluated differently than

```
(a + b) * c
```

4.5. Functions

Functions are the basic building blocks of BackTALK scripts. All scripts have at least one function, named "main()", and script execution begins there. All functions have at least one statement that isn't an INCLUDE or data declaration, and all functions must be declared or defined before they are used in another function. See the section on function declarations below.

The data types of parameters must be declared in the definition line. As many parameters may be declared between the parentheses as needed, separated by commas. Integer parameters are passed as values and changes to them do not affect the variable values in the calling function. String parameters are passed by reference and changes to the string in the called function are permanent. See the section on strings in the previous pages.

4.6. Statements

4.6.1. Assignments

Syntax: <variable> = <expression>;

An assignment gives a value to a variable. The "=" operator assigns the value of the expression on the right to the variable on the left. It will not test the equality of two expressions; the "==" operator is used to test for equality.

4.6.2. Comments

Comments, also called *remarks*, are one of the most important parts of the BackTALK language. A comment is text in a source file that is ignored by the compiler and can be used to explain how each part of the script works. This is very important if the script is intended to be shared with anyone else or if the script ever needs to be modified. Either "--" or "//" signal the beginning of a comment, and it continues to the end of the line. For example:

```
-- This is my first BackTALK program
int     i;          // Declare a global integer variable
string  buf[80];    -- Buffer to store one line of input
```

Comments add nothing to the size of compiled scripts and do not slow their execution. The best programmers use lots of comments.

4.6.3. EXIT Statement

Syntax: EXIT <expression>;

An EXIT statement terminates the script immediately. The expression currently has no effect although it is required to be included. Simply specifying zero, i.e. EXIT 0; is sufficient.

The expression will be used when scripts are given the ability to call other scripts. It must result in an integer, and will be used as a status code to be returned to the script which started the currently running script. The full syntax should be used in current scripts to remain compatible with future versions of BackTALK. Common conventions used for status codes are:

Exit with a status of zero if the script executed successfully, i.e., did not encounter any significant errors.

Exit with a status greater than zero if the script is exiting prematurely due to an error condition. A common convention is that a status of 1 means that non-fatal errors (warnings) were encountered, and greater than one means that fatal errors were encountered.

Negative numbers should only be returned if they correspond to GEMDOS file error codes (see the section on file operations).

4.6.4. Function calls

Syntax: <function name>(parameter list);

A function call consists of the name of the function with the list of parameters the function expects in parentheses following the name. The parameter list can be any number of expressions separated by commas. If no parameters are needed by the

function, the parentheses should be left empty, "()". If the function returns a value, it can be assigned to a variable of the same data type, as in

```
bar = foo();
```

If "foo()" returns an integer value, "bar" must have already been declared as an integer. Return values can also be ignored, so "foo();" is also a valid statement.

Using more parameters than the function needs generally has no bad effects, but using fewer parameters will have unpredictable results.

BackTALK uses a stack structure similar to C. Extra parameters are pushed onto the stack where they cause no problems. If the function goes looking for a parameter that was not passed, however, it will take an unpredictable value from memory.

All functions must be either defined or declared before they are called (see the *Function declarations* section).

4.6.5. Function definitions

Syntax:

FUNCTION <name> (TYPE parm1, TYPE parm2,...) [RETURNS <TYPE>]
[local declarations]
statements
ENDFUNCTION

Every script must have at least one function, main(), where script execution starts. Other functions are defined as needed, and must be either defined or declared (see Function Declarations below) before they are called in any other function. As many parameters as are needed are declared between the parentheses with their respective type, either INT or STRING. If the function will return a value, the RETURNS clause must be used to declare the type.

Following the function name and parameter declarations, any additional variables are declared, and at least one statement is included. To define a function that does nothing, use a null statement (a statement consisting of only the semicolon ";"). The function definition is closed with the ENDFUNCTION clause.

4.6.6. Function declarations

Syntax: FUNCTION <name> [RETURNS <type>];

Functions can be declared in the same way as variables and constants. Declaring a function before it is defined lets the compiler reserve its name and reserve space for its return value (if any) so that the function no longer has to be completely defined before it is used in another function.

4.6.12. WHILE statement

Syntax: **WHILE <expression> DO**
...statements...
ENDWHILE

A WHILE statement repeats execution of a block of statements as long as the test remains TRUE. When BackTALK encounters the WHILE statement, it evaluates the test expression. If the result is TRUE the statements are executed and the expression is evaluated again. The cycle continues until the test expression results in FALSE. String expressions are not allowed - the test must result in an integer. To use a string expression in the test, compare it to the NULL string.

The WHILE statement is a top-tested loop, which means that the statements within the loop may or may not be executed the first time through. The REPEAT statement, above, is a bottom-tested loop, one in which the statements in the loop are always executed at least once.

This example

```
i = 0;
WHILE i < 10 DO
   printf("%d\r\n", i);
   i = i + 1;
ENDWHILE
```

prints the numbers 0 through 9 in the STalker terminal window, while this example

```
WHILE file_gets(string, sizeof(string), the_file) > 0 DO
   port_puts(string);
ENDWHILE
```

sends the contents of a file over the serial port, looping until the end of the file is encountered.

4.7. Built-In Functions

Most of the work done by a BackTALK script will be carried out by the more than 130 built-in functions. These functions are used just like user-defined functions except that their names are not case-sensitive.

4.7.1. A word about file names

One of the features of BackTALK is that it can run as a background task while other GEM programs are running. This is called cooperative multitasking because it requires the tasks that run together to cooperate in sharing the computer. BackTALK and STalker take care of most of the needed cooperation, but the default GEMDOS drive

function, the parentheses should be left empty, "()". If the function returns a value, it can be assigned to a variable of the same data type, as in

```
bar = foo();
```

If "foo()" returns an integer value, "bar" must have already been declared as an integer. Return values can also be ignored, so "foo();" is also a valid statement.

Using more parameters than the function needs generally has no bad effects, but using fewer parameters will have unpredictable results.

BackTALK uses a stack structure similar to C. Extra parameters are pushed onto the stack where they cause no problems. If the function goes looking for a parameter that was not passed, however, it will take an unpredictable value from memory.

All functions must be either defined or declared before they are called (see the *Function declarations* section).

4.6.5. Function definitions

Syntax:

FUNCTION <name> (TYPE parm1, TYPE parm2,...) [RETURNS <TYPE>]
[local declarations]
statements
ENDFUNCTION

Every script must have at least one function, `main()`, where script execution starts. Other functions are defined as needed, and must be either defined or declared (see Function Declarations below) before they are called in any other function. As many parameters as are needed are declared between the parentheses with their respective type, either INT or STRING. If the function will return a value, the RETURNS clause must be used to declare the type.

Following the function name and parameter declarations, any additional variables are declared, and at least one statement is included. To define a function that does nothing, use a null statement (a statement consisting of only the semicolon ";"). The function definition is closed with the ENDFUNCTION clause.

4.6.6. Function declarations

Syntax: FUNCTION <name> [RETURNS <type>];

Functions can be declared in the same way as variables and constants. Declaring a function before it is defined lets the compiler reserve its name and reserve space for its return value (if any) so that the function no longer has to be completely defined before it is used in another function.

4.6.12. WHILE statement

Syntax: **WHILE <expression> DO**
...statements...
ENDWHILE

A WHILE statement repeats execution of a block of statements as long as the test remains TRUE. When BackTALK encounters the WHILE statement, it evaluates the test expression. If the result is TRUE the statements are executed and the expression is evaluated again. The cycle continues until the test expression results in FALSE. String expressions are not allowed - the test must result in an integer. To use a string expression in the test, compare it to the NULL string.

The WHILE statement is a top-tested loop, which means that the statements within the loop may or may not be executed the first time through. The REPEAT statement, above, is a bottom-tested loop, one in which the statements in the loop are always executed at least once.

This example

```
i = 0;
WHILE i < 10 DO
   printf("%d\r\n", i);
   i = i + 1;
ENDWHILE
```

prints the numbers 0 through 9 in the STalker terminal window, while this example

```
WHILE file_gets(string, sizeof(string), the_file) > 0 DO
   port_puts(string);
ENDWHILE
```

sends the contents of a file over the serial port, looping until the end of the file is encountered.

4.7. Built-In Functions

Most of the work done by a BackTALK script will be carried out by the more than 130 built-in functions. These functions are used just like user-defined functions except that their names are not case-sensitive.

4.7.1. A word about file names

One of the features of BackTALK is that it can run as a background task while other GEM programs are running. This is called cooperative multitasking because it requires the tasks that run together to cooperate in sharing the computer. BackTALK and STalker take care of most of the needed cooperation, but the default GEMDOS drive

function, the parentheses should be left empty, "()". If the function returns a value, it can be assigned to a variable of the same data type, as in

```
bar = foo();
```

If "foo()" returns an integer value, "bar" must have already been declared as an integer. Return values can also be ignored, so "foo();" is also a valid statement.

Using more parameters than the function needs generally has no bad effects, but using fewer parameters will have unpredictable results.

BackTALK uses a stack structure similar to C. Extra parameters are pushed onto the stack where they cause no problems. If the function goes looking for a parameter that was not passed, however, it will take an unpredictable value from memory.

All functions must be either defined or declared before they are called (see the *Function declarations* section).

4.6.5. Function definitions

Syntax:

FUNCTION <name> (TYPE parm1, TYPE parm2,...) [RETURNS <TYPE>]
[local declarations]
statements
ENDFUNCTION

Every script must have at least one function, `main()`, where script execution starts. Other functions are defined as needed, and must be either defined or declared (see Function Declarations below) before they are called in any other function. As many parameters as are needed are declared between the parentheses with their respective type, either INT or STRING. If the function will return a value, the RETURNS clause must be used to declare the type.

Following the function name and parameter declarations, any additional variables are declared, and at least one statement is included. To define a function that does nothing, use a null statement (a statement consisting of only the semicolon ";"). The function definition is closed with the ENDFUNCTION clause.

4.6.6. Function declarations

Syntax: FUNCTION <name> [RETURNS <type>];

Functions can be declared in the same way as variables and constants. Declaring a function before it is defined lets the compiler reserve its name and reserve space for its return value (if any) so that the function no longer has to be completely defined before it is used in another function.

The RETURNS clause is only needed if the function will return a value to whatever function calls it, and can only be type INT or STRING.

4.6.7. IF statement

Syntax: **IF <expression> THEN**
...first statements...
[ELSEIF <expression> THEN
...second statements...]
[ELSE
...third statements...]
ENDIF

An IF statement allows the script to decide between executing two or more blocks of statements depending on the result of a test. If the result of the test expression is not zero the first block of statements will be executed and the remaining blocks will be skipped. If the test expression results in zero the first block of statements will be skipped. String expressions are not currently allowed - the test must result in an integer. To use a string expression in the test, compare it to the NULL string. For example:

```
strstr(reply, "Name?") != NULL
```

results in TRUE if "Name?" is found in the reply.

Any number of optional ELSEIF statements can be included. If the ELSE clause is not included, then if all previous test expressions resulted in FALSE, execution will skip to the next statement after ENDIF.

4.6.8. INCLUDE statement

Syntax: INCLUDE "filename"

The INCLUDE statement tells the BackTALK compiler to open and read the file whose name is enclosed in double quotes as if it were a part of the script. The include file, also called a header file, is simply a source file of BackTALK statements that are useful in many BackTALK scripts. The file STALKER.BTH is included with BackTALK and defines constants for use with the built-in functions.

INCLUDE is also a practical way to write a library of custom functions that can be used in many different scripts. Simply write the desired functions and save them in a header file. When writing a script that will use any of the custom functions, INCLUDE the header file before the first reference to any of the custom functions.

This method is less efficient than using a true library, in that all of the code in the header file will be compiled and included in the object file regardless of the functions actually used. However, libraries are not currently supported. This does not apply to the standard STALKER.BTH file, since constant definitions take up no space in the compiled script.

4.6.9. NULL Statement

Syntax: ;

As mentioned above, if a statement is required but no action should be taken, a null statement may be used. A null statement is nothing more than a single semicolon. Do not confuse this with the NULL string, an entirely different concept.

4.6.10. REPEAT statement

Syntax: **REPEAT**
...statements...
UNTIL <expression>;

A REPEAT statement repeats a block of statements until a test expression results in FALSE. When BackTALK encounters the REPEAT statement, it executes the block of statements and then evaluates the test expression. If the result is FALSE the statements are executed again. The cycle continues until the result of the test expression is TRUE. String expressions are not allowed - the test must result in an integer. To use a string expression in the test, compare it to the NULL string.

The REPEAT statement is a bottom-tested loop, which means that the statements within the loop will always be executed at least once on the first time through the loop. The WHILE statement, below, is a top-tested loop, meaning that the statements within the loop may or may not be executed depending on the result of the test expression.

4.6.11. RETURN Statement

Syntax: RETURN [expression];

The RETURN statement is the way that one function in BackTALK returns a result to the function that called it. A typical function call in BackTALK looks like

```
bar = foo();
```

and the function definition of `foo()` must have a RETURN statement with the syntax

```
RETURN [expression];
```

The expression is not needed if the function returns no value to its caller. If used, it must return the data type specified in the function definition, and the variable that the result of the function is assigned to must also be declared as the same data type. Remember that a function can only return an integer or a string pointer.

4.6.12. WHILE statement

Syntax: **WHILE <expression> DO**
...statements...
ENDWHILE

A WHILE statement repeats execution of a block of statements as long as the test remains TRUE. When BackTALK encounters the WHILE statement, it evaluates the test expression. If the result is TRUE the statements are executed and the expression is evaluated again. The cycle continues until the test expression results in FALSE. String expressions are not allowed - the test must result in an integer. To use a string expression in the test, compare it to the NULL string.

The WHILE statement is a top-tested loop, which means that the statements within the loop may or may not be executed the first time through. The REPEAT statement, above, is a bottom-tested loop, one in which the statements in the loop are always executed at least once.

This example

```
i = 0;
WHILE i < 10 DO
   printf("%d\r\n", i);
   i = i + 1;
ENDWHILE
```

prints the numbers 0 through 9 in the STalker terminal window, while this example

```
WHILE file_gets(string, sizeof(string), the_file) > 0 DO
   port_puts(string);
ENDWHILE
```

sends the contents of a file over the serial port, looping until the end of the file is encountered.

4.7. Built-In Functions

Most of the work done by a BackTALK script will be carried out by the more than 130 built-in functions. These functions are used just like user-defined functions except that their names are not case-sensitive.

4.7.1. A word about file names

One of the features of BackTALK is that it can run as a background task while other GEM programs are running. This is called cooperative multitasking because it requires the tasks that run together to cooperate in sharing the computer. BackTALK and STalker take care of most of the needed cooperation, but the default GEMDOS drive

and path can be changed without the knowledge of BackTALK or STalker. When writing scripts using file handling functions, the user should make sure that complete drive and pathnames are supplied, to avoid having file operations happen in strange and unexpected places.

4.7.2. alert()

Syntax: button = alert(STRING alerttext);

Displays a GEM alert box and returns the number of the button selected by the user (an integer). The string `alerttext` consists of three sections each enclosed in square brackets. The format is:

```
"[icon][message][buttons]"
```

The number *icon* is a digit between 0 and 3 and determines the type of icon shown in the alert box. The types are:

0 - no alert icon, text only
1 - a "note" icon (exclamation point)
2 - a "wait" icon (question mark)
3 - a "stop" icon (stop sign)

The *message* portion contains the text displayed by the alert. It may consist of up to five lines each containing up to 30 characters. Lines are separated within the string by "|" characters, for example

```
[This is line one,|this is line two,|and this is line three.]
```

The *buttons* portion contains the text for up to three selector buttons below the text in the box. The total size of the button portion should not exceed 30 characters. The text for each button is separated by a "|" character, for example

```
[First Button|Second Button]
```

The first button will automatically be the "default" button.

4.7.3. appl_id()

Syntax: stalker_id = appl_id();

This function is for use in advanced GEM programming only. This function returns STalker's GEM application identifier for use in other advanced GEM functions.

4.7.4. appl_find()

Syntax: int = appl_find(STRING appl_name);

This function is analagous to the GEM `appl_find()` call. As with the GEM version, the appl_name must be padded with blanks to be exactly 8 characters long. Do NOT include the .PRG/.ACC extension. A value of -1 is returned if the application is not presently running

```
INT neo_cli_id;

neo_cli_id = appl_find("NEO_CLI ");
IF neo_cli_id == -1 THEN
   puts("NeoDesk CLI not present!\r\n");
ELSE
   puts("NeoDesk CLI is present!\r\n");
ENDIF
```

4.7.5. appl_write()

Syntax: success = appl_write(INT appl_id, INT msg_type, INT msg3, INT msg4, INT msg5, INT msg6, INT msg7);

This function is for use in advanced GEM programming only. It sends a message to the application specified by `appl_id` through the GEM message pipe. The `msg_type` indicates the type of GEM message being sent:

Type	Name	Description
10	MN_SELECTED	A menu item is selected
20	WM_REDRAW	A window needs to be redrawn
21	WM_TOPPED	A new active window is selected
22	WM_CLOSED	The close box was selected
23	WM_FULLED	The full box was selected
24	WM_ARROWED	Scroll box or arrows are selected
25	WM_HSLID	Horizontal slider being moved
26	WM_VSLID	Vertical slider being moved
27	WM_SIZED	The size box has been moved
28	WM_MOVED	The move bar has been moved
29	WM_NEWTOP	A new window has become active
40	AC_OPEN	A desk accessory has been selected
41	AC_CLOSE	Desk accessories have been closed

Note: These names are not included in STALKER.BTH.

The numbers `msg3` through `msg7` correspond to words 3 through 7 of the GEM `appl_write()` function. STalker supplies words 0, 1 and 2.

Type	msg3	msg4	msg5	msg6	msg7
10	menu title	menu item			
20	window handle	x coordinate	y coordinate	width	height
21	window handle				
22	window handle				
23	window handle				
24	window handle	action			
25	window handle	position			
26	window handle	position			
27	window handle	x coordinate	y coordinate	width	height
28	window handle	x coordinate	y coordinate	width	height
29	window handle				
40	menuid				
41	menuid				

Note: Further details about GEM messages can be found in various texts on GEM programming, including the Atari Developer's Kit.

4.7.6. ascii_send()

Syntax: success = ascii_send(STRING filename);

Opens the file specified by `filename` and sends the characters to the communication port. If the operation is successful, including opening the file, the function returns TRUE (one). If an error occurs or the upload is aborted with the [Undo] key, the return value will be FALSE (zero). If this function is called while STalker is in ASCII receive mode, it will return immediately with a value of FALSE.

4.7.7. ascii_receive()

Syntax: success = ascii_receive(STRING filename, INT append_if_exists);

The first time the function is called, it creates the disk file specified by `filename` and begins directing characters received from the communication port into the open file. If a file with the same name already exists and `append_if_exists` is TRUE, the new text is added to the file. Otherwise, the old file is destroyed. If the operation, including creating the file, was successful, the function returns TRUE (one). If an error occurs the return value will be FALSE (zero). While the file is receiving characters, all other script functions except `ascii_send()`, `download()`, and `upload()` are available. The disk file will continue to collect characters until a file error occurs, a second `ascii_receive(NULL)` function call is made, or the user presses [Undo] on the keyboard.

Since this operation can inadvertently destroy a file, a good program will use the `file_exists()` function to test for the existence of the file before creating it. (Unless, of course, it doesn't matter whether the file is destroyed.)

4.7.8. atoi()

Syntax: number = atoi(STRING string);

Converts a decimal representation of a number stored in the string to an integer. The function skips leading spaces, \r, and \n characters, taking each subsequent character as a digit in the number until a non-numeric character is found, then returns the value up to that point. See also itoa() and strtoi().

4.7.9. chat_mode()

Syntax: chat_mode();

This function creates a split-screen effect used by STalker for remote talk mode. Outgoing text is displayed in the lower half of the window, and incomgin text in the upper half. Pressing [Shift][Undo] on the local terminal or [Control][C] on the remote terminal terminates talk mode and causes the function to return. No return value.

4.7.10. crc()

Syntax: crc_value = crc(STRING buffer, INT count, INT crc_value)

Returns an integer which is the result of a Cyclic Redundancy Check on the characters in the buffer. Only the lower 16 bits of crc_value are significant. The buffer is not assumed to be null terminated, so the function will process "count" characters in the buffer. See the description of crc32() for further explanation.

4.7.11. crc_update()

Syntax: crc_value = crc_update(INT char, INT crc_value)

Returns the updated Cyclic Redundancy Check based on the old crc_value and the additional character.

4.7.12. crc32()

Syntax: crc_value = crc32(STRING buffer, INT count, INT crc_value)

Returns an integer which is the result of a 32-bit Cyclic Redundancy Check on the characters in the buffer. The buffer is not assumed to be null terminated, so the function will process count characters in the buffer.

All of the CRC functions are used to take a sequence of bytes and compute some number derived from that sequence. The 32 bit versions compute a 32 bit number and the other versions compute a 16 bit number, the data itself is always just a sequence of bytes. The 32 bit version is simply that much more reliable since you've got 65,536

times as many values that the check value can be, giving (in theory) 65,536 times as much reliability. It is possible that two sequences of bytes result in the same CRC, but the mathematics behind CRCs makes this fairly unlikely if the sequences are the same length and are not terribly (over 1K bytes) long.

4.7.13. crc32_update()

Syntax: crc_value = crc32_update(INT char, INT crc_value)

Returns the updated 32-bit Cyclic Redundancy Check based on the old `crc_value` and the additional character.

4.7.14. dial_group()

Syntax: index = dial_group(INT group_no);

Invokes STalker's autodialer, dialing the entries whose `Menu#` field matches `group_no`. `Group_no` must be a binary value between 0 and 9. The function returns FALSE (zero) if any error occurs, including no connection made, otherwise a positive integer indicating the autodialer entry which connected.

4.7.15. dial_load()

Syntax: success = dial_load(STRING filename);

This function loads the autodialer entries from the indicated file ust as if the "Load" button had been selected in the autodialer. Returns zero if the load was successful, non-zero if not.

4.7.16. dial_many()

Syntax: index = dial_many(STRING name1, STRING name2, STRING name2, ..., NULL);

Invokes STalker's autodialer, dialing the entries whose names begin with the *name* strings. The parameters can be literal strings enclosed in double quotes, or the names of previously declared string variable, or any combination. Upper and lower case differences between the passed strings and the STalker autodialer entries are ignored. The function continues dialing until one of the attempts is successful or the number of retries specified in the autodialer is reached. The return value is FALSE (zero) if any error occurred, including no connection made, otherwise a positive integer indicating which of the entries in the parameter list was successful.

For example, if the function call

```
dial_many("mybbs", "yourbbs", "genie", "bix", NULL);
```

returns 3, it indicates that STalker has successfully connected with the "GEnie" entry in the autodialer, including executing the logon sequence (if any). Any auto-execute script linked to the entry will not be executed. If more than one dial entry matches a single *name* string, all matching entries will be dialed in sequence.

4.7.17. dial_one()

Syntax: index = dial_one(STRING name);

Invokes STalker's autodialer, dialing the entry whose name begins with the `name` string. The parameter can be a literal string enclosed in double quotes, or the name of a previously declared string variable. Upper and lower case differences between the passed string and the STalker autodialer entries are ignored. The return value is FALSE (zero) if any error occurred, including no connection made and otherwise a positive number indicating the corresponding autodialer entry. Any auto-execute script linked to the entry will not be executed. If more than one dial entry matches the `name` string, only the first will be dialed.

4.7.18. dial_save()

Syntax: success = dial_save(STRING filename);

This function saves the autodialer entries to the indicated file, just as if the "Save" button had been selected in the autodialer. Returns a zero if the save was successful, non-zero if not.

4.7.19. download()

Syntax: success = download(STRING filename);

Opens the disk file specified by `filename` and begins a download using the currently selected protocol (see the `set_protocol()` function). If the operation is successful, including the creation of the file and completion of the download, the function returns TRUE (one). If an error occurs the return value will be FALSE (zero). If this function is called while STalker is in ASCII receive mode, it will return immediately with a value of FALSE.

STalker will not respond to requests for automatic ZModem download from remote hosts during script execution. If the script needs to respond to an automatic ZModem download request, it should look for the string "*\030B00", which indicates a ZModem download request from the host. This might be useful if the script needed to deal with other messages, e.g. "file not found".

4.7.20. exit_program()

Syntax: exit_program();

This function takes no parameters and returns nothing. If STalker is operating in program mode, it will cause STalker to terminate immediately. If STalker is operating as a desk accessory, the call does nothing and returns immediately. If fact, you can use this to take whatever actions are appropriate in accessory mode by placing statements after the `exit_program()` call.

```
exit_program();
      //  If the script gets here, we must be in ACC mode
exit(0);    // or whatever else is appropriate
```

4.7.21. file_attrib()

Syntax: attributes = file_attrib(STRING pathname, INT mode, INT attribs);

Sets or retrieves the file attributes of the file specified by the `pathname`. If `mode` is 1, the final attributes are set according to the value in `attribs`. Attempting to set an illegal combination of attributes (setting the volume label or subdirectory attributes on a normal file, for example) results in a "access denied error" (-36). Other file errors will return their respective negative codes. If no error occurs, the final file attributes are returned. If `mode` is 0, the file attributes are returned as a bit pattern, and `attribs` is ignored.

The attribute setting can be obtained by adding together the following integers:

Mark	Attribute
0x01	Read Only
0x02	Hidden
0x04	System
0x08	Volume Label
0x10	Subdirectory
0x20	Archive

For example, a hidden file with the archive attribute set would have an attribute of 0x022. Note that the built-in desktop directory routines will display a file unless both the hidden and system attributes are set (NeoDesk does not have this problem). The archive attribute behaves differently in version of TOS before and after 1.4. In version 1.4 and later, the archive bit is set for newly created files and set each time the file is changed, consistent with MS-DOS. In versions before 1.4, the archive attribute is unset for newly created files and unset when the file is changed.

Note: The archive attribute is commonly used by file backup programs (like Diamond Back) to perform incremental backups. When such a utility saves a file to backup storage, it modifies the archive attribute. When a file is changed, the

archive attribute is again modified, allowing the backup utility to identify which files have been changed since the last backup.

This fuction is identical to the GEMDOS `Fattrib()` call.

4.7.22. file_change_dir()

Syntax: success = file_change_dir(STRING path);

Changes the default GEMDOS drive and directory to those indicated by path. Conventional directory notation applies: a letter followed by a colon ":" indicates a drive letter, a leading "\" indicates the path is relative to the root directory of the current drive, no leading "\" indicates the path is relative to the current path. The path need not end with a trailing "\". Returns zero if successful, otherwise a negative error code. Use extreme care when changing the default path, since other applications may be confused by the change.

4.7.23. file_close()

Syntax: status = file_close(INT filehandle);

Closes the indicated file. The integer `filehandle` is the number returned by the `file_open()` or `file_create()` operations. The integer `status` is usually ignored because even if an error occurred, the file is considered closed.

If the return value is negative, it corresponds to the GEMDOS error code.

4.7.24. file_create()

Syntax: filehandle = file_create(STRING filename);

Creates a file with the given file name. If a file with the same name already exists it is destroyed. If the operation is successful, the file will be open and ready to be written to. The integer `filehandle` will contain a non-negative (zero or positive) value that must be used to identify the file in further file operations. If the operation was not successful (the drive or path does not exist, the disk is write-protected, etc.), a negative value will be returned.

Since this operation can inadvertently destroy a file, a good program will use the `file_exists()` function to test for the existence of the file before creating it. (Unless, of course, it doesn't matter whether the file is destroyed.)

4.7.25. file_delete()

Syntax: success = file_delete(STRING filename);

Deletes the file. If the operation is successful, the function returns a value of 0. If not successful, a negative value is returned.

The current drive and directory information should never be relied upon unless obtained immediately before use. Complete pathnames should be used in functions requiring file names wherever possible.

4.7.26. file_dir_first()

Syntax: status = file_dir_first(STRING pathspec, INT attributes, STRING buffer);

Begins a directory search operation. The string `pathspec` is a path specification usually containing 'wildcard' characters, e.g. "C:*.*". The number `attributes` determines the kinds of files selected by the search. The string `buffer` should be declared 24 characters long, and will contain the file information for the first file name found that meets the search criteria. The number `attributes` is a bit mask whose bits have the following meanings:

Bit	Hex Value	Meaning
0	0x01	read-only
1	0x02	hidden
2	0x04	system file (also hidden)
3	0x08	disk volume label
4	0x10	subdirectory name
5	0x20	the archive bit

"Normal" file names, those with no special attributes, are always selected. Setting the other bits to one tells the function to select the other types of file names respectively. If none of these bits are set, i.e., if `attributes` is zero, then volume labels, subdirectories, and hidden or system files are not returned. If the hidden, system, or subdirectory bits are set then files with those attributes will be returned in addition to normal files. If the volume label bit is set, only volume labels will be returned. To build the `attributes` number, the bit-wise OR operation is handy. For example:

```
attribute = 0x02 BITOR 0x10
```

produces a number that will select hidden files and subdirectories in addition to normal files.

The file information is returned in `buffer` as follows:

Offset	Contents
0	The attribute bits of the file.
1-2	The high & low bytes of the file time stamp.
3-4	The high & low bytes of the file date stamp.
5-8	The 4 bytes of the file size (high to low).
9-23	Up to 14 bytes containing the file name.

The time and date bytes are organized identically to those returned by the `get_time()` function, and the same expressions can be used to translate them into

readable character strings. The elements of the `buffer` string, however, are 8-bit bytes and must be combined into an integer to be compatible. The following expressions can be used to do the job:

```
time = (buffer[1] << 8) | buffer[2];
date = (buffer[3] << 8) | buffer[4];
```

The status returned is zero if a file matching the specification and the attributes was found. A non-zero number is returned if no matches are found or if an error occurs.

4.7.27. file_dir_next()

Syntax: `status = file_dir_next(STRING buffer);`

Continues a directory search started with `file_dir_first()`. The directory search must be started with a `file_dir_first()` function. Otherwise, results are unpredictable. If another file meeting the selection criteria established by the `file_dir_first()` function is found, its information is placed in the string "buffer" the same way as the `file_dir_first()` function did, and the status returned is zero. A non-zero number is returned if no further matches are found or if an error occurs.

Here is an example to read and display the names of all of the files in a directory (`C:\*.*` in this example) to the STalker terminal window:

```
string   buffer[24];
int      status;
int      f_attr;
int      f_time;
int      f_date;
int      f_size;
string   f_name;    // Pointer into 'buffer' at file name
int      subdir;
puts("Attr Time Date FileSize FileName\r\n");
status = file_dir_first("c:\\*.*", 0x16, buffer);
while status = 0 do
   f_attr = buffer[0];
   f_time = (buffer[1] << 8) BITOR buffer[2];
   f_date = (buffer[3] << 8) BITOR buffer[4];
   f_size = (buffer[5] << 24) BITOR (buffer[6] << 16)
            (buffer[7] << 8)  BITOR buffer[8];
   f_name = buffer + 9;
   if f_attr BITAND 0x10 then
      subdir = '\\';
   else
      subdir = ' ';
   endif
   printf(" %02x  %04x %04x %04x%04x %s%c\r\n",
      f_attr, f_time, f_date, f_size_high, f_size_low,
```

```
        f_name, subdir);
    status = file_dir_next(buffer);
endwhile
```

4.7.28. file_exists()

Syntax: exists = file_exists(STRING pathname);

Returns TRUE (one) if the specified file already exists, FALSE (zero) if not.

4.7.29. file_getchar()

Syntax: char = file_getchar(INT filehandle);

Fetches the next character from an open file and advances the file pointer by one. The integer `filehandle` is the number returned by the `file_open()` operation. If successful, the fetched character is returned, otherwise a negative value is returned. If the end of the file is reached the value FERR_EOF (defined in STALKER.BTH) is returned. This function accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles.

4.7.30. file_get_dir()

Syntax: pointer = file_get_dir(INT drive_number, STRING current_dir);

Gets the "current" directory on the specified drive from GEMDOS, storing it in the string `current_dir`, which should be long enough to hold complex directory strings. The drive is specified as an integer, with 0 representing the "current" drive, 1 meaning drive "A", 2 meaning drive "B", etc.

The return value is the pointer to the `current_dir` string.

4.7.31. file_get_drive()

Syntax: drive = file_get_drive();

Returns a character representing the current GEMDOS drive, i.e., "A", "B", "C", etc.

4.7.32. file_gets()

Syntax: length = file_gets(STRING buffer, INT maximum, INT filehandle);

Reads characters from an open file and places them in the string `buffer`. It reads until encountering a linefeed ("\n") or until (`maximum - 1`) characters have been read. The number `filehandle` is the value returned by the `file_open()` operation. The returned value is the actual number of characters read (integer) or a negative value if a file error occurred. If the end of the file was encountered during the read, the length

returned will be less than `maximum`, possibly zero. It accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles.

A common and effective way to program this function is to use the `sizeof()` function to insert the actual size of the buffer string as `maximum`. This allows the buffer size to be changed without searching for and changing each individual use.

When `file_gets()` returns, the string `buffer` will be null-terminated, hence the maximum number of characters that will fit in the string is one less than the number `maximum`.

4.7.33. file_instat()

Syntax: `status = file_instat(INT file_handle);`

Returns TRUE (one) if the specified device has a character ready to be read, FALSE (zero) if not. The function accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles. It always returns FALSE for the printer and TRUE for disk file handles.

4.7.34. file_open()

Syntax: `filehandle = file_open(STRING filename, INT mode);`

Opens an existing file (`filename`) in either read/write or read-only mode (`mode`) and places the file pointer at the beginning of the file. A mode of FO_READ (0) indicates read-only, and a mode of FO_READWRITE (1) indicates read/write. If the operation is successful, the integer `filehandle` will contain a positive value that must be used to identify the file in further file operations. If the operation was not successful (the file could not be found, the file or disk is set to "read-only" and the file open mode was read/write, etc.), a negative value will be returned.

The number of files that can be open at the same time depends on the size of the BackTALK script buffer in the *STalker Preferences* settings.

Buffer Size	Number of Files
1K	0 (file operations not permitted)
2K	1
3K	2
5K	3
8K	5
16K	8
>16K	10

STalker manages file operations while other programs are executing by maintaining a small buffer for each open BackTALK file, and only doing disk operations when necessary. If a BackTALK script opens a file on a floppy disk and the user removes or

switches the disk, the next time STalker needs to access the file it will get a "file not found" error and will return this to the script. A well-written script will be able to deal with this error, otherwise users of the script should be cautioned against changing floppy disks while STalker is using files on them.

Opening a file that is already open will result in another file handle with its own buffer and file pointer. It might be useful to have two channels to the same file in read-only mode, but if one of the channels is read/write, the read-only channel may return incorrect data because of delays in writing data in the other channel. Use of multiple file handles for the same physical file in anything but read-only mode is not recommended.

4.7.35. file_outstat()

Syntax: status = file_outstat(INT file_handle);

Returns TRUE (one) if the specified device is ready to accept characters, FALSE (zero) if not. The function accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles. It always returns TRUE for disk file handles and the console.

4.7.36. file_printf()

Syntax: count = file_printf(INT file_handle, STRING format, arguments);

Sends a formatted character string to the device specified by `file_handle`. The formatting is identical to the `printf()` function. It accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles, and returns the actual number of characters output.

4.7.37. file_putchar()

Syntax: char = file_putchar(INT filehandle, INT char);

Writes the lower byte of the integer `char` to an open file. The number `file_handle` is the number returned by the `file_open()` or `file_create()` operation, and the file must have been opened in read/write mode. If the write is successful, the value of `char` is returned. A negative value is returned if there was a file error. It accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles.

4.7.38. file_puts()

Syntax: length = file_puts(STRING buffer, INT file_handle);

Writes the characters in the string buffer to an open file. The number file_handle is the number returned by the file_open() or file_create() operation, and the file must have been opened in read/write mode. If the write is successful, the number returned will be the number of characters written. If a file error occurs, a negative value is returned and the number of characters actually written to the file is not known. It accepts the special file handles for FILE_CONSOLE (-1), FILE_PORT (-2) and FILE_PRINTER (-3) as well as disk file handles.

The buffer must be null-terminated, but the null is not written to the file and is not counted in the return value. Therefore, a successful write will return strlen(buffer).

4.7.39. file_read()

Syntax: length = file_read(INT file_handle, STRING buffer, INT maximum);

Reads characters from an open file and places them in the string buffer. It reads until maximum characters have been read. The number file_handle is the value returned by the file_open() operation. The returned value is the actual number of characters read (integer) or a negative value if a file error occurred. If the end of the file was encountered during the read, the length returned will be less than maximum, possibly zero. The file_read() function is used to fill the buffer string with binary data that might contain zeroes, and so an ASCII 0 would not necessarily mark the end of the string.

A common and effective way to program this function is to use the sizeof() function to insert the actual size of the buffer string. This allows the buffer size to be changed without searching for and changing each individual use.

4.7.40. file_rename()

Syntax: success = file_rename(STRING orig_name, STRING new_name);

Changes the name of a disk file from orig_name to new_name. Returns 0 if successful, or a negative value if not.

4.7.41. file_seek()

Syntax: offset = file_seek(INT filehandle, INT mode, INT offset);

Moves the file pointer, or the position where the next character will be read or written, of an open file by the number of bytes specified by `offset`. Positive values of `offset` move the pointer forward in the file (toward the end) and negative values move it backward (toward the beginning). The `filehandle` is the number returned by the `file_open()` operation. The values of `mode` may be:

Value	STALKER.BTH Name
0	FSEEK_FROM_BOF
1	FSEEK_FROM_HERE
2	FSEEK_FROM_EOF

Value 0: Seek from the beginning of the file (offset should be positive or zero)
Value 1: Seek from the current file position (offset may be positive, negative, or zero)
Value 2: Seek backward from the end of the file (offset should be negative or zero).

The value returned is the new file pointer position in bytes from the beginning of the file, or a negative value if an error occurs.

4.7.42. file_write()

Syntax: length = file_write(INT filehandle, STRING buffer, INT maximum);

Writes characters from the string `buffer` to an open file. The number `filehandle` is the number returned from the `file_open()` or `file_create()` operations, and the file must have been opened in read/write mode. The string `buffer` is not assumed to be null-terminated, and the function will continue to write until the number of characters specified by `maximum` are written, or a file error occurs. The `file_write()` function is used to send binary data that might contain an ASCII 0 to a disk file. If the operation is successful, the number returned is the actual number of characters written. If a file error occurs, a negative value will be returned and the number of characters actually written to the file is not known.

4.7.43. fkey_load()

Syntax: success = fkey_load(STRING filename);

Loads a STalker function key file just as if the "Load" button had been selected in the *Function Key Settings* dialog box. Returns zero if the load was successful, or non-zero if not. If the load is not successful, the previous function key entries are left intact.

4.7.44. fkey_save()

Syntax: success = fkey_save(STRING filename);

Saves a STalker function key file just as if the "Save" button had been selected in the *Function Key Settings* dialog box. Returns zero if the save was successful, or non-zero if not.

4.7.45. fsel()

Syntax: confirm = fsel(STRING def_path, STRING def_name, STRING pathname, STRING prompt);

Calls the GEM file selector and allows the user to choose a directory and filename. The string `def_path` is the default directory and file mask as displayed in the "Directory" line of the file selector, including the drive letter. The string `def_name` is the default file name as it appears in the "Selection" line of the file selector. The string `prompt` is a message that appears with the selector to remind the user what the file selection is for.

If the user exits the selector with the "Ok" button, the function returns TRUE (one), the complete pathname of the file the user selected is stored in the string `pathname`. If the user exits the selector with the "Cancel" button, the function returns FALSE (zero). The default strings are updated in either case.

Insure that the `def_path` and `pathname` strings are long enough to contain long directory chains. The `def_name` string should be declared at least 13 characters long (12 for the filename and one for the null). The maximum length of the `prompt` string is 30 characters; it must not be NULL.

This function displays the prompt string regardless of what version of TOS the machine is using. For versions of TOS below 1.4, the prompt will be displayed in its own box above the file selector. For TOS 1.4 and above, the prompt appears in the top of the file selector itself. Remember that backslashes have special meaning in character strings. To enter a backslash, it must be doubled, e.g. the directory path "`A:\*.*`" should be entered "`A:\\*.*`". See the `printf()` function for more information.

4.7.46. getchar()

Syntax: char = getchar();

Fetches a character from the keyboard buffer. If a character is not waiting in the buffer, meaning that a key hasn't been pressed, the function will wait until a key is pressed. The character is not echoed to the screen. The ASCII value (integer) of the key is returned. If the key has no ASCII value (e.g. a function key), the function returns ASCII 0.

4.7.47. getkey()

Syntax: scan_code = getkey();

Fetches a character from the keyboard buffer, similar to `getchar()`. If a character is not waiting in the buffer, meaning that a key hasn't been pressed, the function will wait until a key is pressed. The character is not echoed to the screen. The lower 8 bits of the returned integer contains the ASCII value of the key pressed, exactly like `getchar()`. The upper 8 bits contain the specific scan code of the key, a unique identifier based on the location of the key on the keyboard. Numeric keys on the keypad and on the top row return the same ASCII value, but different scan codes. Function keys, the [Help], [Insert], [Clr/Home], [Backspace] and [Undo] keys return an ASCII value of zero, but a non-zero scan code. A table of scan codes is provided in Appendix B.

The expression

```
getkey() & 0x00FF
```

will provide the ASCII value (lower 8 bits) of the integer returned by `getkey()` and is functionally equivalent to `get_char()`, while:

```
(getkey() & 0xFF00) >> 8
```

provides the scan code (upper 8 bits).

Note: Some keystrokes are intercepted by STalker (e.g. [Alternate][Tab]) or automatically processed by the system (e.g. [Alternate][Help]) and may not be available to the script.

4.7.48. gets()

Syntax: length = gets(STRING buffer, INT maximum);

Accepts characters from the keyboard and stores them in the string `buffer`. The function reads until a carriage return ("\r") or a linefeed ("\n") is typed, or (`maximum - 1`) characters are read. The characters are echoed to the screen during input, and backspace and delete keys work to edit in the expected way. The actual number of characters read (integer) is returned.

A common and effective way to program this function is to use the `sizeof()` function to insert the actual size of the buffer string as `maximum`. This allows the buffer size to be changed without searching for and changing each individual use.

The string `buffer` will be null-terminated, hence the maximum number of characters that will fit in the string is one less than the number maximum. The length returned does not include the trailing null.

4.7.49. get_baud()

Syntax: rate = get_baud();

Returns the current baud rate of the communications port (integer).

4.7.50. get_capture()

Syntax: cap_status = get_capture();

Returns TRUE (one) if STalker is currently in capture mode, FALSE (zero) otherwise.

4.7.51. get_duplex()

Syntax: duplex_mode = get_duplex();

Returns an integer corresponding to STalker's current echo mode. The values are:

Value	Mode	STALKER.BTH Constant Name
0	Full	DUPLEX_FULL
1	Half	DUPLEX_ECHO
2	Local	DUPLEX_LOCAL

4.7.52. get_emulation()

Syntax: emul_mode = get_emulation();

Returns an integer corresponding to STalker's current emulation mode. The values are:

Value	Mode	STALKER.BTH Constant Name
0	VT52	EMUL_VT52
1	VT100	EMUL_VT100
2	PCANSI	EMUL_PCANSI

4.7.53. get_flow_control()

Syntax: flow_mode = get_flow_control();

Returns an integer corresponding to the current flow control mode. The values are:

Value	Mode	STALKER.BTH Constant Name
0	None	FLOW_NONE
1	X-On/X-Off	FLOW_XON
2	RTS	FLOW_RTS
3	Both	FLOW_BOTH

4.7.54. get_int_param()

Syntax: parameter = get_int_param(INT type, INT number);

Returns the value of the STalker integer parameter indicated by `type` and `number`. See `set_int_param()` for a complete description of available parameters.

4.7.55. get_parity()

Syntax: parity_mode = get_parity();

Returns an integer corresponding to the current parity setting of the communications port. The values are:

Value	Mode	STALKER.BTH Constant Name
0	None	PARITY_NONE
1	Even	PARITY_EVEN
2	Odd	PARITY_ODD
3	Mark(1)	PARITY_MARK
4	Space(0)	PARITY_SPACE

4.7.56. get_port()

Syntax: port = get_port();

Returns an integer corresponding to the device that STalker is using as the communications port. The values are:

Value	Device	STALKER.BTH Constant Name
0	Standard RS232	PORT_RS232
1	Standard MIDI	PORT_MIDI
2	Standard Parallel	PORT_PRINTER
3	TT "Serial 1"	PORT_SERIAL1
4	TT/MegaSTe "Serial 2"	PORT_SERIAL2
5	TT/MegaSTe "Modem 2"	PORT_MODEM2
6	Double Click Port 1	PORT_DCP1
7	Double Click Port 2	PORT_DCP2
8	MegaTalk Port A	PORT_MTKA
9	MegaTalk Port B	PORT_MTKB
10	MT C-Shell ttyp0	PORT_PTY0
11	MT C-Shell ttyp1	PORT_PTY1
12	MT C-Shell ttyp2	PORT_PTY2
13	MT C-Shell ttyp3	PORT_PTY3

4.7.57. get_printer()

Syntax: print_status = get_printer();

Returns TRUE (one) if STalker is set to echo text to the printer, FALSE (zero) otherwise.

4.7.58. get_protocol()

Syntax: protocol = get_protocol();

Returns an integer corresponding to the current file transfer protocol. The values are:

Value	Protocol	STALKER.BTH Constant Name
0	Xmodem	PROTO_XMODEM
1	Xmodem/1K	PROTO_XMODEM1K
2	Ymodem	PROTO_YMODEM
3	Ymodem/G	PROTO_YMODEMG
4	Zmodem	PROTO_ZMODEM

4.7.59. get_stop_bits()

Syntax: stop_bits = get_stop_bits();

Returns the number of stop bits per character used in the current port parameters: 1, 2, or 3, with 3 indicating 1.5 stop bits.

4.7.60. get_string_param()

Syntax: pointer = get_string_param(INT type, INT number, STRING buffer, INT max);

Stores the STalker string parameter corresponding to `type` and `number` in the string `buffer`. The integer `max` is the maximum number of characters the function will place in the string, and should usually be `sizeof(buffer)`. It returns the pointer to the buffer. See `set_string_param()` for a complete description of available parameters.

A common and effective way to program this function is to use the `sizeof()` function to insert the actual size of the buffer string as `max`. This allows the buffer size to be changed without searching for and changing each individual use.

4.7.61. get_time()

Syntax: date_time = get_time();

Returns an integer containing the current system year, month, day, hour, minute, and seconds in binary format as follows:

Upper 16 bits from high to low:	YYYY YYYM MMMD DDDD
Lower 16 bits from high to low:	HHHH HMMM MMMS SSSS

The "Y" bits are the number of years since 1980, a number between 0 and 119. The "M" bits are the month, between 1 and 12, and the "D" bits are the day of the month, between 1 and 31. The "H" bits are the hours between 0 and 23, the "M" bits are the minutes between 0 and 59, and the "S" bits are the seconds divided by 2 between 0 and 29. The most convenient way to format and display the result of this function is with the `strftime()` function. However, these expressions allow manipulation of the data bits directly:

If the result of `get_time()` is assigned to `time`, the expression

```
(time >> 11) & 0x1F
```

will return the hour in 24-hour format, the expression

```
(time >> 5) & 0x3F
```

will return the minutes, and

```
2 * (time & 0x1F)
```

will return the seconds.

This function uses the `gettime()` system call and is limited to its 2-second accuracy. For more accurate timing see the `timer_()` functions.

If the result of `get_time()` is assigned to `date`, the expression

```
(date >> 25) & 0x7F) + 1980
```

will return the year in four-digit format, the expression

```
(date >> 21) & 0xF
```

will return the month, and

```
(date >> 16) & 0x1F
```

will return the day of the month.

This function is identical to the XBIOS `Gettime()` call.

4.7.62. get_word_length()

Syntax: word_length = get_word_length();

Returns the number of data bits per character used in the current port parameters, which ranges between 5 and 8.

4.7.63. global_address()

Syntax: int = global_address(STRING local_address);

This routine converts a BackTALK address (whicih is relative to the start of the script execution buffer) to a global physical system address suitable for passing to another GEM application. Example:

```
int string_address;

// Pass a null-termianted string in two halves using
// words 3 and 4 of the GEM message.
string_address = global_address("My dog has fleas.");
appl_write(neo_cli_id, 9876, string_address,
   string_address >> 16, 0, 0, 0);
```

4.7.64. instat()

Syntax: status = instat();

Checks the status of the console (keyboard) and returns non-zero if it has a character ready to be read, FALSE (zero) if not.

4.7.65. isalnum()

Syntax: status = isalnum(INT character);

Returns non-zero if the character is in the range 0-9, A-Z or a-z, otherwise returns FALSE (zero).

4.7.66. isalpha()

Syntax: status = isalpha(INT character);

Returns non-zero if the character is in the range A-Z or a-z, otherwise returns FALSE (zero).

4.7.67. iscntrl()

```
Syntax: status = iscntrl(INT character);
```

Returns non-zero if the character is in the range ASCII 0-31 or ASCII 127 (delete), otherwise returns FALSE (zero).

4.7.68. isdigit()

```
Syntax: status = isdigit(INT character);
```

Returns non-zero if the character is in the range 0-9, otherwise returns FALSE (zero).

4.7.69. isgraph()

```
Syntax: status = isgraph(INT character);
```

Returns non-zero if the character is a "visible" ASCII character, ASCII 33-255 except ASCII 127, otherwise returns FALSE (zero).

4.7.70. islower()

```
Syntax: status = islower(INT character);
```

Returns non-zero if the character is in the range a-z, otherwise returns FALSE (zero).

4.7.71. isodigit()

```
Syntax: status = isodigit(INT character);
```

Returns non-zero if the character is in the range 0-7 (octal digit), otherwise returns FALSE (zero).

4.7.72. isprint()

```
Syntax: status = isprint(INT character);
```

Returns non-zero if the character is a "printable" ASCII character, ASCII 32-255 except ASCII 127, otherwise returns FALSE (zero).

4.7.73. ispunct()

```
Syntax: status = ispunct(INT character);
```

Returns non-zero if the character is a "punctuation" character, ASCII 33-126 except ASCII 32, 0-9, A-Z and a-z, otherwise returns FALSE (zero).

4.7.74. isspace()

Syntax: status = isspace(INT character);

Returns non-zero if the character is a "whitespace" character, ASCII 9-13 or ASCII 32, otherwise returns FALSE (zero).

4.7.75. isupper()

Syntax: status = isupper(INT character);

Returns non-zero if the character is in the range A-Z, otherwise returns FALSE (zero).

4.7.76. isxdigit()

Syntax: status = isxdigit(INT character);

Returns non-zero if the character is in the range 0-9, A-F or a-f (hexadecimal digits), otherwise returns FALSE (zero).

4.7.77. itoa()

Syntax: pointer = itoa(INT number, STRING buffer, INT base);

Converts an integer `number` into a string of ASCII digits and stores them in `buffer`. The `base` is the number base the integer should be converted to, for example 2 (binary), 8 (octal), 16 (hexadecimal), or 10 (decimal). Any number from 2 to 36 can be used for the number base. Returns the pointer to the buffer string. See also `atoi()` and `strtoi()`.

4.7.78. memory_read()

Syntax: value = memory_read(INT location);

Returns the value of the byte stored in the indicated memory location. *This function is for advanced programming only!* Improper use of this function can cause system crashes! The memory access occurs in 680x0 supervisor mode.

4.7.79. memory_write()

Syntax: memory_write(INT location, INT value);

Inserts the low order 8 bits of the indicated value into the byte at the indicated memory location. *This function is for advanced programming only!* Improper use of this function can cause data corruption and system crashes! The memory access occurs in 680x0 supervisor mode.

4.7.80. outstat()

Syntax: status = outstat();

Always returns TRUE. (Checks the status of the console and returns TRUE (one) if it is ready to accept a character, FALSE (zero) if not.)

4.7.81. play_sound()

Syntax: play_sound(INT sound);

Tells STalker to play one of the built-in melodies.

Sound #	Melody	STALKER.BTH Name
0	Dragnet	SOUND_DRAGNET
1	Charge	SOUND_CHARGE
2	We're In The Money	SOUND_THE_MONEY
3	Beethoven's Fifth	SOUND_FIFTH
4	Shave & A Haircut	SOUND_SHAVE_HCUT
5	Do You Want To Know A Secret?	SOUND_SECRET

Any other value is ignored. Any other sound generated by the computer, like the "keyclick" sound will interrupt the melody. If the script is running in the background, the playback may be interrupted by any foreground-generated sound.

4.7.82. port_break()

Syntax: port_break();

Sends a communications break of approximately 250 milliseconds to the communications port. If the port does not support a break, nothing is sent.

4.7.83. port_carrier()

Syntax: carrier_status = port_carrier();

Returns TRUE (one) if the carrier detect signal is active on the communications port or if the port does not support carrier detect, otherwise FALSE. This function is not affected by the status of the "Modem sets DCD properly" setting in the *Modem Settings* dialog.

4.7.84. port_connected()

Syntax: connect_status = port_connected();

Returns TRUE (one) if STalker's internal flag indicates it is connected to a host, otherwise it returns FALSE (zero). The flag is set when the autodialer successfully dials a number, and is cleared when STalker first starts up or after the hang-up command is

issued. The title of the STalker window also indicates the status of the flag: if the title is "Off-Line", the flag is cleared.

4.7.85. port_drop_dtr()

Syntax: `port_drop_dtr();`

Interrupts the Data Terminal Ready (DTR) signal on the communications port for approximately 750 milliseconds and raises it again. This is an effective way to hang up for configurations that support the DTR signal. If the communications port or the modem does not support the DTR signal this function does nothing. This function is not affected by the status of the "Drop DTR" setting in the *Modem Settings* dialog.

4.7.86. prot_dtr()

Syntax: `port_dtr(INT raise_dtr);`

This routine lets you explicitly control the DTR signal. If raise_dtr is TRUE, the DTR signal is asserted, otherwise it is lowered. Example:

```
//  Do the same thing as port_dropt_dtr()

port_dtr(FALSE);      // drop DTR
wait(7, FALSE);       // wait a bit
port_dtr(TRUE);       // bring it back up again
```

4.7.87. port_getchar()

Syntax: `char = port_getchar();`

Returns the next available character (integer) from the communications port. If there is no character waiting, the function will wait until one arrives.

4.7.88. port_gets()

Syntax: `length = port_gets(STRING buffer, INT maximum)`

Reads characters from the communications port and places them in the string `buffer`. It reads until encountering a linefeed ("\n") or until (`maximum - 1`) characters have been read. The returned value is the actual number of characters read (integer).

A common and effective way to program this function is to use the `sizeof()` function to insert the actual size of the buffer string as `maximum`. This allows the buffer size to be changed without searching for and changing each individual use.

The string `buffer` will be null-terminated, hence the maximum number of characters that will fit in the string is one less than the number `maximum`.

4.7.89. port_hang_up()

Syntax: port_hang_up();

This function is equivalent to pressing [Alternate][H] in STalker's interactive mode. It reads STalker's *Modem Configuration* and either sends the hangup string to the modem or interrupts the RS232 DTR signal, depending on the setting found. The result is to hang up the phone.

4.7.90. port_instat()

Syntax: port_status = port_instat();

Returns TRUE (one) if a character is waiting at the communications port, otherwise returns FALSE (zero). Use this function before `port_getchar()` to determine if it will have to wait for a character.

4.7.91. port_outstat()

Syntax: ready_status = port_outstat();

Returns TRUE (one) if there is room for at least one character in the output buffer of the communications port, otherwise FALSE (zero). Use this function before `port_putchar()` to determine if it will have to wait for the buffer to clear.

4.7.92. port_printf()

Syntax: count = port_printf(STRING format, arguments);

Sends a formatted character string to the current communications port. The formatting is identical to the `printf()` function. Returns the number of characters sent.

4.7.93. port_putchar()

Syntax: char = port_putchar(INT char);

Sends the lower 8 bits of the integer `char` to the communications port. If the output buffer is full and can not accept more input, this function will wait until there is room in the buffer for the character. Returns the character value.

4.7.94. port_puts()

Syntax: counts = port_puts(STRING buffer);

Sends each character in the string `buffer` to the communications port. If the output buffer is full, the function will wait until there is room. The actual number of characters output is returned. The string is not echoed to the screen except by the host, and will therefore not be visible if STalker is in LOCAL or ECHO mode.

4.7.95. port_send()

Syntax: pointer = port_send(STRING buffer, INT as_function_key);

Sends each character in the string `buffer` to the communications port, similar to the `port_puts()` function. All characters sent will be echoed to the screen, when STalker is in LOCAL or ECHO mode. If "`as_function_key`" is FALSE (zero) the characters will be sent without interpretation. If "`as_function_key`" is TRUE (one), the characters are interpreted as a function key macro. That means that the special delay characters (ASCII 0xAB, ½) will cause a pause and chaining characters (ASCII 0x9F, *f*) will execute other function keys as if they were sent by a function key. BackTALK scripts called will not be run. Returns the pointer to the string.

4.7.96. port_write()

Syntax: count = port_write(STRING buffer, INT maximum);

Writes characters from the string `buffer` to the communications port. The string `buffer` is not assumed to be null-terminated, and the function will continue to send until the number of characters specified by `maximum` are sent. The `port_write()` function is used to send binary data that might contain an ASCII 0 to the communications port. Returns the actual number of characters output.

4.7.97. printer_outstat()

Syntax: status = printer_outstat();

Returns TRUE (one) if the printer is ready to accept characters, or FALSE (zero) if not. If printing is attempted when the printer is not ready, GEMDOS will wait 30 seconds and display an alert which halts further processing until the user acknowledges it. Use this function to check the printer status before printing.

4.7.98. printer_printf()

Syntax: count = printer_printf(STRING format, arguments);

Sends a formatted character string to the printer. The formatting is identical to the `printf()` function. The value returned is the number of characters sent. Use the `printer_outstat()` function to check the printer status before printing.

4.7.99. printer_putchar()

Syntax: count = printer_putchar(INT char);

Sends the lower 8 bits of the integer `char` to the printer. No return value. If the printer is not ready, GEMDOS will wait 30 seconds, display an error alert, and wait for

the user to acknowledge. Use the printer_outstat() function to check the printer status before printing. Returns the actual number of characters output.

4.7.100. printer_puts()

Syntax: count = printer_puts(STRING string);

Sends the null-terminated string to the printer. The print head is not automatically advanced to the next line unless the string contains carriage return and linefeed characters. If the printer is not ready, GEMDOS will wait 30 seconds, display an error alert. Use the printer_outstat() function to check the printer status before printing. Returns the actual number of characters output.

4.7.101. printf()

Syntax: count = printf(STRING format, arguments);

Formats the list of arguments according to the commands contained in the format string and displays the string to the screen. The format string consists of two things: ordinary characters and conversion commands. The ordinary characters are simply copied to the screen without interpretation. The conversion commands take the next argument in the parameter list, format it, and then display the result. For example:

```
printf("This is the number ten in hexadecimal: 0x%x\r\n", 10);
```

will print

```
This is the number ten in hexadecimal: 0xa
```

The "%" character starts the conversion command. The conversion type, in this case "x", tells the function how to treat the argument. The possible conversions are:

Character	Action
d	Display in decimal form
x	Display in hexadecimal form
X	Display in hexadecimal, using uppercase letters
c	Display the argument as a single ASCII character
s	The argument is a string

The conversion can be modified by placing modifiers between the "%" and the conversion command character:

```
%<flag><min_width><.max_width>
```

The flag can be either "-" to specify left alignment or "0" to specify leading zeroes on numeric conversions.

The field min_width specifies the minimum width of the output field.

The `max_width` specifies the maximum number of characters that should be printed from a string - it has no effect on numeric conversions.

The field width and precision may also be specified as "*", which causes them to use the next integer argument in the parameter list.

For example: `printf("%*d", 20, 5)` will print the number 5 in a field of width 20.

A number, e.g.: "%10s", tells the function to print the string in a space at least 10 characters wide, or wider if the string is longer. If the number starts with a zero, i.e. "%05d", the function pads the display with zeroes.

If the number is followed by a decimal and another number, e.g.: "%10.8s", the function will not print more than that number of characters in the field, even if the string is longer. In this example, the field will always have two spaces at the beginning.

If the "%" character is followed by a minus sign ("-"), the argument is left-justified within the field; padding spaces or zeroes will be at the end of the field instead of the beginning.

Example:

If `name` contains "Arthur", the expression

```
printf("Dear %s", name);
printf(",\r\n");
```

prints "Dear Arthur", and advances to the next line.

4.7.102. putchar()

Syntax: `char = putchar(INT char);`

Displays the character represented by the lower 8 bits of `char` to the screen at the current cursor position. Returns the character value.

4.7.103. puts()

Syntax: `count = puts(STRING buffer);`

Displays the characters in the string `buffer` to the screen. The cursor is not automatically advanced to the next line unless the string contains carriage return and linefeed characters. Returns the actual number of characters output.

4.7.104. run_program()

```
Syntax: exit_code = run_program(STRING pathname, STRING
        cmd_line, INT redirect);
```

Executes the external program specified by `pathname` passing it a command line specified by `cmd_line`. Normally only non-GEM applications TOS and TTP file types, may be run using this command. If `redirect` is TRUE (non-zero), STalker will attempt to redirect the input and output of the program through the current communications port, allowing remote access. Redirection may not be successful depending on the selected port.

If redirection fails, the return value is `-1` and the program is not executed. Otherwise, the return value is the value returned from the GEMDOS `Pexec()` call used to execute the program. This can include the exit code of the program or the GEMDOS error code if a file error was encountered on loading the program file, among other things.

When running a TOS program, it clears the screen, turns on the TOS cursor, runs the program, then turns off the TOS cursor, and restores the GEM screen.

If you are using STalker as a *NeoDesk Accessory*, this function will have NeoDesk run the program for you. There are several advantages to this: higher compatibility and any type of program can be executed (GEM, APP, GTP, TOS, TTP).

4.7.105. set_baud()

```
Syntax: rate = set_baud(INT baud);
```

Sets the communications port to the rate specified by `baud`. If the indicated rate is not available, the port is set to the next lower available rate. Returns the actual rate set.

4.7.106. set_capture()

```
Syntax: cap_status = set_capture(INT cap_state);
```

Sets the capture status specified by `cap_state`; 0 for capture off, 1 for capture on. Requires STeno to be present in the system. Returns an integer corresponding to the final capture state.

4.7.107. set_duplex()

```
Syntax: int = set_duplex(INT requested_mode);
```

This function sets the echo mode. It takes one integer parameter (the constants DUPLEX_FULL, DUPLEX,ECHO, and DUPLEX_LOCAL are defined for the `get_duplex()` call) and sets the current duplex mode. The resulting mode is

returned, which may differ from the requested mode if the requested mode is invalid. Example:

```
int old_duplex;

old_duplex = set_duplex(DUPLEX_FULL);     // get old mode
....perform actions...
set_duplex(old_duplex);                   //  reset
```

4.7.108. set_emulation()

Syntax: emul_mode = set_emulation(INT emulation_type);

Sets STalker's emulation mode according to `emulation_type`. Returns an integer corresponding to the final emulation mode. The values are

Value	Mode	STALKER.BTH Constant Name
0	VT52	EMUL_VT52
1	VT100	EMUL_VT100
2	PCANSI	EMUL_PCANSI

4.7.109. set_flow_control()

Syntax: flow_mode = set_flow_control(INT flow_mode);

Sets the communications port to the flow control mode specified by `flow_mode`. Returns the actual flow mode set. Possible values are:

Value	Mode	STALKER.BTH Constant Name
0	None	FLOW_NONE
1	X-On/X-Off	FLOW_XON
2	RTS	FLOW_RTS
3	Both	FLOW_BOTH

4.7.110. set_int_param()

Syntax: parameter = set_int_param(INT type, INT number, INT value);

Sets the STalker integer parameter indicated by `type` and `number` to `value`. Returns the value of the parameter actually set. Note that entry lists are numbered starting with zero. For the types for which the `number` paramerer is ignored, there must still be a value in that parameter position. Passing a zero is sufficient.

Allowed values of `type` and `number` and their corresponding parameters are:

Type Values	STALKER.BTH Name	Description
1	DIALER_COST_PARAM	#1
2	DIALER_GROUP_PARAM	#2

3	DIALER_LIMIT_PARAM	#3
7	DIALER_LOGON_PARAM	#4
16	MDM_USE_DCD_PARAM	#5
17	MDM_USE_DTR_PARAM	#6
18	MDM_DELAY_PARAM	#7
22	XFER_PARAM	#8
50	MISC_PARAM	#9

#1 The autodialer $/Hr field, including cents, but without a decimal ($30.25/Hr would be entered as 3025) from 0 to 9999. The number indicates which dialer entry, from 0 to 29.

#2 The autodialer Menu# field, from 0 to 9. Use -1 to clear the field. The number indicates which dialer entry, from 0 to 29.

#3 The autodialer Limit field, including cents but without a decimal ($50.00 would be entered as 5000) from 0 to 9999. The number indicates which dialer entry, from 0 to 29.

#4 The autologon flags attached to an autodialer entry. The number is a combination of the autodialer entry in the lower 16 bits with the following in the upper 16 bits:

DIALER_LG_HIDDEN = 0x000

The upper 16 bits consist of a number indicating which response string, from 0 to 7. The returned value will be TRUE or FALSE.

For example, to get the state of the 3rd "hidden" flag (DIALER_LG_HIDDEN + 2) of the 7th (6 + ...) dialer entry use:

```
hidden = get_int_param(DIALER_LOGON_PARAM, 6
+((DIALER_LG_HIDDEN + 2) <<16));
```

#5 The "modem sets DCD properly" flag, which returns TRUE or FALSE. The number is ignored.

#6 The "drop DTR" flag, which returns TRUE or FALSE. The number is ignored.

#7 The delay between characters when dialing the modem, in 1/100 of a second. The number is ignored.

#8 The file transfer parameters. The number can have the following values:

Value	STALKER.BTH Name	Description
0	ASCII_ADD_LF	TRUE/FALSE, add if after received CRs in a "receive"
3	ASCII_LINE_WAIT	Wait this long (in 10ths of a second) after sending line

4	ASCII_STRIP_LF	TRUE/FALSE, strip LFs from files during a "send"
5	ASCII_USE_CTRL_Z	TRUE/FALSE, treat a ^Z character as "end-of-file"
10	BATCH_SUBDIRECTS	How to traverse subdirectories during a batch upload: 0, 1, or 2
11	BATCH_FILENAMES	How to send file names in batch uploads: 0, 1, or 2
15	XMODEM_LOOSE_TIMING	TRUE/FALSE, relax timing for X/YModem transfers
16	XMODEM_SIZE_BYTE	TRUE/FALSE, pad partial XModem 128 blocks with a size byte
20	ZMODEM_ESCAPE-CTRL	TRUE/FALSE, encode all control characters in ZModem transfer
21	ZMODEM_RCV_OPTION	0 to 7, See Table A
22	ZMODEM_RCV_MODE	0 to 2, See Table B
25	ZMODEM_SEND_OPTION	0 to 7, See Table A
26	ZMODEM_SEND_MODE	0 to 2, See Table B
27	ZMODEM_SEND_CRC16	TRUE/FALSE, Use 16 bit CRCs in ZModem uploads

TABLE A- Allowed values for ZMODEM_RCV/SEND_OPTION

STALKER.BTH Name	Value	Description
ZOPT_NONE	0	No option specified
ZOPT_NEWL	1	Transfer if destination newer or longer
ZOPT_CRC	2	Compute a CRC on source and destination, transfer if they differ
ZOPT_APND	3	Append the source file to the destination
ZOPT_CLOB	4	Overwrite the destination
ZOPT_NEW	5	Transfer if source is newer
ZOPT_DIFF	6	Transfer if sizes or time-stamps differ
ZOPT_PROT	7	Don't overwrite the destination

TABLE B - Allowed values for ZMODEM_RCV/SEND_MODE

STALKER.BTH Name	Value	Description
ZCONV_AUTO	0	Download: use sender's choice
ZCONV_BINARY	1	Transfer file without performing any sort of conversion
ZCONV_TEXT	2	Transfer file in text mode

#9 Various settings as indicated by the number below:

Number	STALKER.BTH Name	Description
0	SHOW_CONTROL_CHARS	TRUE/FALSE, show control characters option.
1	SCRAP_DRIVE	The Clipboard Drive, "A", "B", "C", etc
2	GMT_OFFSET	Offset in minutes (+/-) from GMT
3	TYPEAHEAD_STYLE	ON will use current ASCII upload settings, OFF sends text at full speed
666		Enables/Disables the processing of Clr/Home sequences

4.7.111. set_parity()

Syntax: parity_mode = set_parity(INT parity_mode);

Sets the communications port to the parity specified by parity_mode. Returns the actual parity mode set. Possible values are:

Value	Mode	STALKER.BTH Constant Name
0	None	PARITY_NONE
1	Even	PARITY_EVEN
2	Odd	PARITY_ODD
3	Mark	PARITY_MARK
4	Space	PARITY_SPACE

4.7.112. set_string_param()

Syntax: pointer = set_string_param(INT type, INT number, STRING buffer);

Sets the STalker string parameter indicated by type and number to the contents of the string buffer. Insure that buffer is no longer than the allowed length of the string parameter, or unexpected results will occur. Returns the pointer to the buffer, or NULL if there was an error such as passing incorrect type and number.

The special function key control characters can be included in the buffer when setting function key strings. They are:

Function	Char	Hex Value
Pause	½	0xAB
Chain	ƒ	0x9F
Script	§	0xDD

For the types for which the number parameter is ignored, there must still be a value in that parameter position. Passing a zero is sufficient. Note that entry lists are numbered starting with zero.

Allowed values of type and number and their corresponding parameters are:

Type	STALKER.BTH Name	Description
0	FKEY_PARAM	#1
4	DIALER_NAME_PARAM	#2
5	DIALER_NUMBER_PARAM	#3
6	DIALER_SCRIPT_PARAM	#4
7	DIALER_LOGON_PARAM	#5
10	MDM_INIT_PARAM	#6
11	MDM_PREFIX_PARAM	#7
12	MDM_SUFFIX_PARAM	#8
13	MDM_HANGUP_PARAM	#9
14	MDM_CONNECT_PARAM	#10
15	MDM_FAILURE_PARAM	#11
20	PATH_PARAM	#12
21	REMOTE_PARAM	#13
22	XFER_PARAM	#14

#1 Function key strings. The number indicates which key, from 0 to 19.

#2 Autodialer BBS/System name. The number indicates which entry, from 0 to 29.

#3 Autodialer BBS/System telephone number. The number indicates which entry, from 0 to 29.

#4 The name of the autoexecute BackTALK script attached to an autodialer enty, .BTK file extension not included. The number indicates which entry, from 0 to 29.

#5 The autologon strings attached to an autodialer entry. The number is made up of three parts. The most significant two bytes indicate whether the string is a prompt or a response and which string in the prompt/response list is being referred to. They are constructed by adding either DIALER_LG_PROMPT(0x100) or DIALER_LG_RESPONSE(0x200) to a number from 0 to 7 which indicates which string from the list is to be modified. The least signficant two bytes indicate the autodialer entry number, from 0 to 29. For example, to set the 3rd prompt string of the 7th dialer entry to "Name?" use:

```
set_string_param(DIALER_LOGON_PARAM,
(((DIALER_LG_PROMPT + 2) << 16) + 6),"Name? ");
```

Or, more generally, to set the xth response string of the yth entry to "JDoe\n":

```
set_string_param(DIALER_LOGON_PARAM,
```

```
((DIALER_LG_RESPONSE + (x - 1)) << 16) + (y - 1)),
"JDoe\n");
```

#6 The modem Initialization string. The number is ignored.

#7 The modem Prefix string. The number is ignored.

#8 The modem Suffix string. The number is ignored.

#9 The modem Hangup string (sent by [Alternate][H] and port_hang_up() when Drop DTR is not selected). The number is ignored.

#10 The modem Connect strings. The number selects which one, beginning at 0.

#11 The modem Failure strings. The number selects which one, beginning at 0.

#12 A GEMDOS path specification. The number indicates which path:

Number	STALKER.BTH Name	Description
0	PATH_DOWNLOAD	Download path
1	PATH_SCRIPT	BackTALK script path
2	PATH_GENERAL	General purpose path
3	PATH_SCRAP	Clipboard path (GEM AES scrp_read()/scrp_write())

#13 Remote mode parameters. The number options are as follows:

Number	STALKER.BTH Name	Description
0	REMOTE_PASSWORD	Remote mode password
1	REMOTE_CLI_PASSWORD	Password to run CLI
2	REMOTE_CLI_NAME	Full pathname of the CLI

#14 ASCII transfer parameters. The number options are:

Number	STALKER.BTH Name	Description
1	ASCII_BLANK_EXPAND	Expand blank lines to this string in a send
2	ASCII_HOST_PROMPT	Wait for this string after each line in a send

4.7.113. set_port()

Syntax: port = set_port(INT port);

This function is used to choose the device used by STalker as the communications port. It returns the actual communications port set. Possible ports and their corresponding values of port are:

Value	Device	STALKER.BTH Constant Name
0	Standard RS232	PORT_RS232
1	Standard MIDI	PORT_MIDI
2	Standard Parallel	PORT_PRINTER
3	TT "Serial 1"	PORT_SERIAL1
4	TT/MegaSTe "Serial 2"	PORT_SERIAL2
5	TT/MegaSTe "Modem 2"	PORT_MODEM2
6	Double Click Port 1	PORT_DCP1
7	Double Click Port 2	PORT_DCP2
8	MegaTalk Port A	PORT_MTKA
9	MegaTalk Port B	PORT_MTKB
10	MT C-Shell ttyp0	PORT_PTY0
11	MT C-Shell ttyp1	PORT_PTY1
12	MT C-Shell ttyp2	PORT_PTY2
13	MT C-Shell ttyp3	PORT_PTY3

4.7.114. set_printer()

Syntax: print_status = set_printer(INT state);

When `state` is TRUE (one), this function sets STalker to echo characters written to the terminal window to the printer. If `state` is FALSE (zero), printer echo is turned off. Returns the final state, TRUE or FALSE.

4.7.115. set_protocol()

Syntax: protocol = set_protocol(INT protocol);

Sets STalker's file transfer protocol. Possible values of `protocol` and the corresponding modes are:

Value	Protocol	STalker+.BTH Constant Name
0	Xmodem	PROTO_XMODEM
1	Xmodem/1K	PROTO_XMODEM1K
2	Ymodem	PROTO_YMODEM
3	Ymodem/G	PROTO_YMODEMG
4	Zmodem	PROTO_ZMODEM

4.7.116. set_stop_bits()

Syntax: stop_bits = set_stop_bits(INT stop_bits);

Sets the communications port to use the specified number of stop bits, either 1, 2, or 3, where 3 indicates 1.5 stop bits. Other values are ignored. Returns the actual stop bits value set.

4.7.117. set_time()

Syntax: `set_time(date_time);`

This is the converse of the `get_time()` call and takes one parameter which should have the desired date and time packed into the same format returned by `get_time()`. Internally, this function sets both the XBIOS and GEMDOS time and date.

4.7.118. set_word_length()

Syntax: `word_length = set_word_length(INT word_length);`

Sets the communications port to use the specified number of data bits in each "word". `word_length` may be 5, 6, 7, or 8. Other values are ignored. Returns the actual word length set.

4.7.119. show_message()

Syntax: `pointer= show_message(STRING message, INT temp);`

Displays a message following the service name in the title bar of the STalker window. If `message` is NULL, then any previous message is removed. If `temp` is TRUE (non-zero), the message will automatically disappear after approximately five seconds, otherwise the message will remain until another message replaces it or it is removed. Returns the pointer to the message string.

4.7.120. sizeof()

Syntax: `size = sizeof(name);`

Returns an integer containing the size of the object (integer, string, or string pointer) named as the argument. Returns a value of 4 for integers and string pointers, and returns the declared size of strings. (The function `strlen()` returns the number of characters in the string up to the first null.)

Note: `Sizeof()` is actually a compiler directive to substitute the size of the specified identifier for the expression. The size is not evaluated at runtime.

4.7.121. sprintf()

Syntax: `count = sprintf(STRING output, STRING format, arguments);`

This function is similar to `printf()` in that it produces a string arranged according to the formatting commands in the `format` string. Instead of displaying it on the screen it is placed in the `output` string. Insure that the `output` string is declared to be large enough to contain the result of this function. Refer to the explanation of

printf() for further details. Returns the number of characters placed in the output string.

4.7.122. steno_append()

Syntax: success = steno_append(STRING buffer);

Adds the characters in buffer to the end of the STeno buffer (regardless of the current cursor position in STeno). Returns TRUE (one) if the operation was successful, FALSE (zero) if not (STeno not available, out if buffer space, etc.) Since STeno uses its own line-ending characters, the incoming string should not contain carriage return ("\r") or linefeed ("\n") characters unless the ^M and ^J characters are desired in the text.

4.7.123. steno_id()

Syntax: steno_id = steno_id();

This function is for use in advanced GEM programming only. This function returns the GEM application identification (integer) of the STeno accessory that STalker is communicating with, for use in other advanced GEM functions. If STeno is not available it returns -1.

4.7.124. strcat()

Syntax: pointer = strcat(STRING destination, STRING source);

Catenates, or connects, the source string to the end of the destination string, and returns the pointer to the destination string. Since strings are simply arrays of bytes, arithmetic addition of two strings, e.g. "Hello," and " world!", will not result in "Hello, world!". The strcat() function moves the elements of the elements of the source string into the destination string so that they appear in the correct order. The destination string must have been declared large enough to contain the new string or unpredictable results may occur.

4.7.125. strchr()

Syntax: pointer = strchr(STRING buffer, INT char);

Returns the pointer to the first occurrence of char in the string buffer. If the character is not found before a null is reached, the function returns NULL.

4.7.126. strcmp()

Syntax: result = strcmp(STRING first, STRING second);

Compares the first and second strings character by character. If they are identical in all characters, the function returns 0. If they are not identical, the values of the first pair of characters that differ are compared. If the value of the character in the first string is larger than the corresponding character in the second string, the result is positive. If it is smaller, the result is negative.

4.7.127. strcpy()

Syntax: pointer = strcpy(STRING destination, STRING source);

Copies the contents of the source string to the destination string and returns the pointer to the destination string. Since strings are simply arrays of bytes, assigning one string to another, e.g. greeting = "Hello" will not place the characters "Hello" into the string greeting. The destination string must have been declared large enough to hold the source or unpredictable results may occur.

4.7.128. strftime()

Syntax: count = strftime(STRING buffer, INT max, STRING format, INT time);

Similar to sprintf(), this formats the date/time value in time according to the commands in format and stores the output in buffer. Non-command characters in the format string are written directly to the buffer. At most, max characters are written. Returns the number of characters written to buffer excluding the terminating null. A common and effective way to program this function is to use the sizeof() function to insert the actual size of the buffer string as max. This allows the buffer size to be changed without searching for and changing each individual use.

Command	Result
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	date and time (dd/mm/yy hh:mm:ss)
%d	day of the month (1-31)
%H	hour (24 hour clock, 0-23)
%I	hour (12 hour clock, 1-12)
%j	day of the year (1-366)
%m	Numeric month (1-12)
%M	minute (00-59)
%p	AM/PM
%S	second (00-59)

```
%w          numeric weekday (0-6, Sunday is 0)
%W          numeric week of year (0-52)
%x          date (dd/mm/yy)
%X          time (hh:mm:ss)
%y          year without century (0-99)
%Y          year with century
%%          the '%' character
```

4.7.129. strlen()

Syntax: length = strlen(STRING buffer);

Returns an integer value equal to the number of characters in the string `buffer`, excluding the trailing null.

4.7.130. strlwr()

Syntax: pointer = strlwr(STRING buffer);

Converts characters in the `buffer` in the range A-Z to lower case. Other characters are unaffected. Returns the pointer to the `buffer`.

4.7.131. strncat()

Syntax: pointer = strncat(STRING destination, STRING source, INT maximum);

Similar to `strcat()`, this function will append the `source` string to the `destination` string, but will append no more than the `maximum` number of characters.

Note: The resulting string will not be null-terminated if the source string is longer than `maximum`.

A common and effective way to program this function is to use `sizeof(destination) - strlen(destination)` to calculate the available space in the `destination` string as `maximum`.

4.7.132. strncmp()

Syntax: result = strncmp(STRING first, STRING second, INT maximum);

Compares the `first` and `second` strings character by character. If the first `maximum` characters are identical, the function returns 0. If they are not identical, the values of the first pair of characters that differ are compared. If the value of the character in the `first` string is larger than the corresponding character in the `second` string, the result is positive. If it is smaller, the result is negative.

4.7.133. strncpy()

Syntax: pointer = strncpy(STRING destination, STRING source, INT maximum);

Similar to `strcpy()`, this function will copy the characters from the `source` string into the `destination` string, but will copy no more than the `maximum` number of characters.

Note: The resulting string will not be null-terminated if the `source` string is longer than `maximum`.

A common and effective way to program this function is to use `sizeof(destination) - 1` to calculate the available space in the `destination` string as `maximum`.

4.7.134. strrchr()

Syntax: pointer = strrchr(STRING buffer, INT char);

Similar to `strchr()`, returns a pointer to the last occurrence of `char` in the string `buffer`. If `char` is not found NULL is returned.

4.7.135. strstr()

Syntax: pointer = strstr(STRING search_in STRING search_for);

Returns a pointer to the first occurrence of the `search_for` string in the `search_in` string. If `search_for` is not found, returns NULL.

4.7.136. strtoi()

Syntax: integer = strtoi(STRING digits, INT base);

Returns an integer represented by the digits in the input string. The `base` is the number base in which the digits are written, 2 for binary, 8 for octal, 16 for hexadecimal, etc. Zero or any number between 2 and 36 may be used as the base. The function skips leading spaces, `\r`, and `\n` characters, taking each subsequent character as a digit in the number until a non-numeric character is found, then returns the value up to that point.

If `base` is 0, the function examines the first one or two digits of the string to determine the number base. If the first two characters are "0x" or "0X" followed by a hexadecimal digit (0-9, a-f, A-F), the function assumes hexadecimal notation. If the first character is a "0" (zero) followed by an octal digit (0-7), the function assumes the input string is in octal notation. If the first character is a decimal digit (1-9), the function assumes decimal notation. See also `atoi()` and `itoa()`.

4.7.137. strupr()

Syntax: pointer = strupr(STRING buffer);

Converts all characters in `buffer` in the range a-z to upper case. Other characters are unaffected. Returns a pointer to `buffer`.

4.7.138. timer_check()

Syntax: expired = timer_check(INT timer_name);

Checks the timer described by `timer_name` and returns TRUE (one) if it is expired, otherwise returns FALSE (zero). The timer must have already been initialized by a `timer_set()` function call with a positive time-out value.

The `timer_()` functions use the 200Hz system timer and are suitable for instances where the two-second accuracy of the `get_date()` function is insufficient. The accuracy of these calls depends on the speed of the machine, with typical values between .1 and .5 seconds.

4.7.139. timer_elapsed()

Syntax: time = timer_elapsed(INT timer_name);

Returns the time in tenths of a second since the timer was initialized using the `timer_set()` function. The timer must have been set with a zero time-out value.

The `timer_()` functions use the 200Hz system timer and are suitable for instances where the two-second accuracy of the `get_date()` function is insufficient. The accuracy of these calls depends on the speed of the machine, with typical values between .1 and .5 seconds.

4.7.140. timer_set()

Syntax: timer_name = timer_set(INT time_out);

Sets a timer for use in subsequent `timer_check()` and `timer_elapsed()` functions. Any number of timers may be defined and running at the same time. The `time-out` is the length of time in tenths of a second until the `timer_check()` function will indicate the timer is expired. If `time_out` is set to 0, the timer will count up continuously for use with the `timer_elapsed()` function. Returns an integer that must be used to identify the timer in subsequent `timer_check()` or `timer_elapsed()` calls. The return value has no other significance and no other operations should be attempted with it.

The `timer_()` functions use the 200Hz system timer and are suitable for instances where the two-second accuracy of the `get_time()` function is insufficient. The

accuracy of these calls depends on the speed of the machine, with typical values between .1 and .5 seconds.

4.7.141. tolower()

Syntax: letter = tolower(INT char);

Converts the given single character to lowercase. If the character is not a valid uppercase character (A to Z) it will return the same character given to it.

4.7.142. toupper()

Syntax: letter = toupper(INT char);

Converts the given single character to uppercase. If the character is not a valid lowercase character (a to z) it will return the same character given to it.

4.7.143. upload()

Syntax: success = upload(STRING filename);

Opens the file specified by `filename` and sends it to the communication port using the currently selected protocol (see the `set_protocol()` function). If the operation is successful or if it was aborted with the [Undo] key the function returns TRUE (one) and if not successful it returns FALSE (zero). If this function is called while STalker is in ASCII receive mode, it will return immediately with a value of FALSE.

4.7.144. wait()

Syntax: success = wait(INT time, INT read_port);

This function delays script execution for `time` tenths of a second, then returns TRUE. Pressing [Undo] during the delay aborts the delay and the function returns FALSE. If `read_port` is FALSE, incoming characters during the delay period are held in the communications buffer. If the buffer fills, STalker will attempt to use flow control to stop the data flow. If flow control is disabled or if the host does not support flow control, characters may be lost. If `read_port` is TRUE, incoming characters are accepted by STalker.

4.7.145. wait_for()

Syntax: success = wait_for(STRING prompt, INT time_out);

Pauses while monitoring the communications port for the `prompt` string. If the `prompt` is received within `time_out` tenths of a second, the function returns TRUE (one). If the time out period expires before the `prompt` is received, the function returns FALSE (zero). If the [Undo] key is pressed during the pause, the function will abort and

return a value of FALSE. If the time_out is set to zero, the function will not time out and will wait for the prompt until it is received or the [Undo] key is pressed. Incoming characters are accepted by STalker.

4.7.146. wait_until()

Syntax: success = wait_until(INT year, INT month, INT day, INT hour, INT minute);

Pauses script execution until the computer's system clock matches the date and time given. Year should be expressed in four-digit format (e.g. 1990) and hour should be expressed in 24 hour format, i.e. 10pm is 22 hours. The function returns TRUE (one) when the appointed time and date is reached, or if the date and time specified was in the past. If the [Undo] key is pressed during the wait, the function aborts and returns a value of FALSE (zero). Incoming characters are accepted by STalker.

Negative values have special significance to wait_until(): using -1 in any argument tells the function to ignore that part of the date and time when waiting. This is useful in constructing routines that run at regular intervals, i.e. every day at 02:00 or every July 4th. For example:

```
wait_until(-1,-1,-1,23,30);
```

Will either wait until 23:30 for the current year/month/day if the time is before 23:30, or 23:30 for tomorrow if the time is later than 23:30. Using any other negative number in an argument tells the function to substitute the current clock value in that place. This is useful when timing routines to run later the same day. For example:

```
wait_until(-2, -2, -2, 10, 47);
```

Will wait until 10:47 on the current year/month/day. If the current time is past 10:47 the function will return immediately.

4.7.147. write_script()

Syntax: error = write_script(INT file_handle, INT end_key, INT thresh_typed, INT thresh_wait, INT prompt_length);

This function watches data from the host and the user and builds a script containing a sequence of wait_for() and port_send() commands. Included is the script file WRITESCR.BTS which does the necessary stuff before calling write_script(). Pressing [Control][Z] will terminate WRITESCR.BTK.

Note: The scripts generated by this command are rather basic. File downloading and similar items are not recorded. Function keys are referenced by their number. Some tweaking will be in order. See the WRITESCR.BTS file for an example of its usage.

5. Appendix A - ASCII Table

Decimal	Hex	Char	Key	Definition
0	0x00	NUL	ctrl-@	NUL character
1	0x01	SOH	ctrl-A	start of header (up arrow)
2	0x02	STX	ctrl-B	start of text (down arrow)
3	0x03	ETX	ctrl-C	end of text (right arrow)
4	0x04	EOT	ctrl-D	end of transmission (left arrow)
5	0x05	ENQ	ctrl-E	enquiry (close box)
6	0x06	ACK	ctrl-F	positive acknowledge (resize box)
7	0x07	BEL	ctrl-G	ring bell (full box)
8	0x08	BS	ctrl-H	backspace (check mark)
9	0x09	HT	ctrl-I	horizontal tab (clock)
10	0x0A	LF	ctrl-J	line feed (bell)
11	0x0B	VT	ctrl-K	vertical tab (musical note)
12	0x0C	FF	ctrl-L	form feed (FF)
13	0x0D	CR	ctrl-M	carriage return (CR)
14	0x0E	SO	ctrl-N	shift out (left Fuji)
15	0x0F	SI	ctrl-O	shift in (right Fuji)
16	0x10	DLE	ctrl-P	data link escape (digital 0)
17	0x11	DC1	ctrl-Q	device control 1 (XON) (digital 1)
18	0x12	DC2	ctrl-R	device control 2 (tape on) (digital 2)
19	0x13	DC3	ctrl-S	device control 3 (XOFF) (digital 3)
20	0x14	DC4	ctrl-T	device control 4 (tape off) (digital 4)
21	0x15	NAK	ctrl-U	negative acknowledge (digital 5)
22	0x16	SYN	ctrl-V	synchronize (digital 6)
23	0x17	ETB	ctrl-W	end of transmission block (digital 7)
24	0x18	CAN	ctrl-X	cancel (digital 8)
25	0x19	EM	ctrl-Y	end of medium (digital 9)
26	0x1A	SUB	ctrl-Z	substitute (schwa)
27	0x1B	ESC	ctrl-[	escape (ES)
28	0x1C	FS	ctrl-\	form separator (Bob 1)
29	0x1D	GS	ctrl-]	group separator (Bob 2)
30	0x1E	RS	ctrl-^	record separator (Bob 3)
31	0x1F	US	ctrl-_	unit separator (Bob 4)
32	0x20	SP		space
33	0x21	!		exclamation mark
34	0x22	"		quotation mark
35	0x23	#		pound sign
36	0x24	$		dollar sign
37	0x25	%		percent sign
38	0x26	&		ampersand
39	0x27	'		apostrophe
40	0x28	(		left parenthesis
41	0x29	)		right parenthesis
42	0x2A	*		asterisk
43	0x2B	+		plus sign
44	0x2C	,		comma
45	0x2D	-		minus sign (hyphen)
46	0x2E	.		period
47	0x2F	/		virgule (slash)
48	0x30	0		zero
49	0x31	1		one

Decimal	Hex	Char	Key	Definition
50	0x32	2		two
51	0x33	3		three
52	0x34	4		four
53	0x35	5		five
54	0x36	6		six
55	0x37	7		seven
56	0x38	8		eight
57	0x39	9		nine
58	0x3A	:		colon
59	0x3B	;		semicolon
60	0x3C	<		less than
61	0x3D	=		equal sign
62	0x3E	>		greater than
63	0x3F	?		question mark
64	0x40	@		at sign
65	0x41	A		
66	0x42	B		
67	0x43	C		
68	0x44	D		
69	0x45	E		
70	0x46	F		
71	0x47	G		
72	0x48	H		
73	0x49	I		
74	0x4A	J		
75	0x4B	K		
76	0x4C	L		
77	0x4D	M		
78	0x4E	N		
79	0x4F	O		
80	0x50	P		
81	0x51	Q		
82	0x52	R		
83	0x53	S		
84	0x54	T		
85	0x55	U		
86	0x56	V		
87	0x57	W		
88	0x58	X		
89	0x59	Y		
90	0x5A	Z		
91	0x5B	[		left bracket
92	0x5C	\		backslash
93	0x5D	]		right bracket
94	0x5E	^		circumflex
95	0x5F	_		underscore
96	0x60	`		grave
97	0x61	a		
98	0x62	b		
99	0x63	c		
100	0x64	d		
101	0x65	e		
102	0x66	f		
103	0x67	g		
104	0x68	h		
105	0x69	i		
106	0x6A	j		
107	0x6B	k		
108	0x6C	l		
109	0x6D	m		

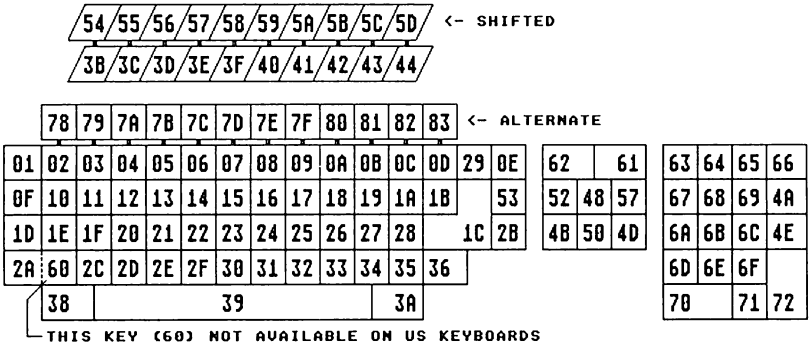
Decimal	Hex	Char	Key	Definition
110	0x6E	n		
111	0x6F	o		
112	0x70	p		
113	0x71	q		
114	0x72	r		
115	0x73	s		
116	0x74	t		
117	0x75	u		
118	0x76	v		
119	0x77	w		
120	0x78	x		
121	0x79	y		
122	0x7A	z		
123	0x7B	{		left brace
124	0x7C	|		vertical bar
125	0x7D	}		right brace
126	0x7E	~		tilde
127	0x7F	D		delete character
128	0x80	Ç		C cedilla
129	0x81	ü		u umlaut
130	0x82	é		e acute
131	0x83	â		a circumflex
132	0x84	ä		a umlaut
133	0x85	à		a grave
134	0x86	å		a ring
135	0x87	ç		c cedilla
136	0x88	ê		e circumflex
137	0x89	ë		e umlaut
138	0x8A	è		e grave
139	0x8B	ï		i umlaut
140	0x8C	î		i circumflex
141	0x8D	ì		i grave
142	0x8E	Ä		A umlaut
143	0x8F	Å		A ring
144	0x90	É		E acute
145	0x91	æ		a e ligature
146	0x92	Æ		A E ligature
147	0x93	ô		o circumflex
148	0x94	ö		o umlaut
149	0x95	ò		o grave
150	0x96	û		u circumflex
151	0x97	ù		u grave
152	0x98	ÿ		y umlaut
153	0x99	Ö		O umlaut
154	0x9A	Ü		U umlaut
155	0x9B	¢		cent symbol
156	0x9C	£		British pound symbol
157	0x9D	¥		Yen symbol
158	0x9E	ß		German sharp-S
159	0x9F	¦		function f
160	0xA0	á		a acute
161	0xA1	í		i acute
162	0xA2	ó		o acute
163	0xA3	ú		u acute
164	0xA4	ñ		n tilde
165	0xA5	Ñ		N tilde
166	0xA6	a		feminine ordinal indicator
167	0xA7	o		masculine ordinal indicator
168	0xA8	¿		upside-down question mark
169	0xA9			or sign

Decimal	Hex	Char	Key	Definition
170	0xAA			not sign
171	0xAB	½		1/2 fraction
172	0xAC	¼		1/4 fraction
173	0xAD	¡		upside-down exclamation point
174	0xAE	«		double less than symbol
175	0xAF	»		double greater than symbol
176	0xB0	ã		a tilde
177	0xB1	õ		o tilde
178	0xB2	Ø		O slash
179	0xB3	ø		o slash
180	0xB4	œ		o e ligature
181	0xB5	Œ		O E ligature
182	0xB6	À		A grave
183	0xB7	Ã		A tilde
184	0xB8	Õ		O tilde
185	0xB9	¨		umlaut (dieresis)
186	0xBA	´		acute accent
187	0xBB	†		daggar (footnote)
188	0xBC	¶		paragraph symbol
189	0xBD	©		copyright symbol
190	0xBE	®		registered trademark symbol
191	0xBF	™		trademark symbol
192	0xC0			ij ligature
193	0xC1			IJ ligature
194	0xC2			Hebrew aleph
195	0xC3			Hebrew beth
196	0xC4			Hebrew gimel
197	0xC5			Hebrew daleth
198	0xC6			Hebrew he
199	0xC7			Hebrew waw
200	0xC8			Hebrew zayin
201	0xC9			Hebrew heth
202	0xCA			Hebrew teth
203	0xCB			Hebrew yod
204	0xCC			Hebrew lkaph
205	0xCD			Hebrew lamed
206	0xCE			Hebrew mem
207	0xCF			Hebrew nun
208	0xD0			Hebrew samekh
209	0xD1			Hebrew ayin
210	0xD2			Hebrew pe
211	0xD3			Hebrew sadhe
212	0xD4			Hebrew qoph
213	0xD5			Hebrew resh
214	0xD6			Hebrew sin
215	0xD7			Hebrew taw
216	0xD8			Hebrew nun
217	0xD9			Hebrew kaph
218	0xDA			Hebrew mem
219	0xDB			Hebrew pe
220	0xDC			Hebrew sadhe
221	0xDD	§		section symbol
222	0xDE	^		low circumflex
223	0xDF	∞		infinity
224	0xE0	α		alpha
225	0xE1	β		beta
226	0xE2	Γ		upper gamma
227	0xE3	π		lower pi
228	0xE4	Σ		upper sigma
229	0xE5	σ		lower sigma

Decimal	Hex	Char	Key	Definition
230	0xE6	µ		lower mu (micro)
231	0xE7	τ		lower tau
232	0xE8	Φ		upper phi
233	0xE9	Θ		upper theta
234	0xEA	Ω		upper omega
235	0xEB	δ		lower delta
236	0xEC	ϕ		lower phi
237	0xED	ϕ		lower phi
238	0xEE	ε		lower epsilon
239	0xEF	∩		intersection
240	0xF0	≡		three horizontal lines
241	0xF1	±		plus/minus sign
242	0xF2	≥		greater than or equal to
243	0xF3	≤		less than or equal to
244	0xF4	⌠		integral sign 1
245	0xF5	⌡		integral sign 2
246	0xF6	÷		division sign
247	0xF7	≈		approx. equal to
248	0xF8	°		degree sign
249	0xF9	∙		middle dot
250	0xFA	·		low dot
251	0xFB	√		square root
252	0xFC	-		middle line
253	0xFD	²		superscript 2
254	0xFE	³		superscript 3
255	0xFF	¯		macron (overline)

6. Appendix B - Keyboard Scancodes

These are the scan codes returned by the Atari keyboard, in hexadecimal. Note that some keys produce different scan codes when used with the [Shift] or [Alternate] keys.



7. Appendix C - Source Code Formatting Conventions

Source code formatting is unquestionably an art, and is largely a matter of personal preference. The objective is to make the text easy to read so that it can be interpreted by anyone looking at the code. A few conventions are used by the majority of programmers, however.

1. Indent logic structures and function definitions. Indenting serves to physically set relatively independent parts of the program apart from the rest of the text. For example:

```
IF expression THEN
    ...then portion
ELSE
    ...else portion
ENDIF

REPEAT
    ...loop body
UNTIL expression

WHILE expression DO
    ...loop body
ENDWHILE

FUNCTION name(int param_1, string param_2) RETURNS INT
    INT     local_1
    STRING  local_2[20];
    ...body of function
ENDFUNCTION
```

2. Place only one statement on a line. Unlike BASIC, in which readability is often improved by linking statements together on a line, BackTALK, like C, is best read with one statement per line.

3. Line up variable names in data declarations in the same screen column. Look at the example function definition above. The declarations for the local variables line up the variable names so that it is easier for the eye to scan the list.

4. Line up comments well to the right of statements.

5. Long function definitions with many parameters are often more readable when each parameter is placed on one line indented from the function call. For example:

```
FUNCTION name(
    INT param_1,        // description of param_1
    INT param_2,        // description of param_2
    ) RETURNS INT       // description of return value
    ... etc
```

   This also allows a comment for each of the parameters.

6. Use upper and lower case consistently. This is one of the least agreed-upon techniques of formatting, but consistent use allows the eye to quickly identify different parts of the program. These examples use upper case for keywords and lower case for everything else, but other schemes may be used. Some programmers capitalize variable names or function names (but not both) and some use case to distinguish between global and local variables.

7. Put a space on either side of operators, for example:

```
foo = bar();
```

   In general, adding more "white space" to the text makes it easier to read.

8. Appendix D - BackTALK Operators

8.1. Binary Operators

Operator	Name	Category
+	addition	arithmetic
-	subtraction	arithmetic
*	multiplication	arithmetic
/	division	arithmetic
%	remainder	arithmetic
&	bit-wise AND	arithmetic
|	bit-wise OR	arithmetic
^	bit-wise exclusive OR	arithmetic
&&	logical AND	logical
||	logical OR	logical
>	greater than	logical
>=	greater than or equal to	logical
<	less than	logical
<=	less than or equal to	logical
==	equal to	logical
!=	not equal to	logical
>>	bit-shift right	arithmetic
<<	bit-shift left	arithmetic

8.2. Unary Operators

Operator	Name	Category
-	negative	arithmetic
~	bit-wise NOT	arithmetic
!	logical NOT	logical

8.3. Operator Precedence

Order	Operators	
1	- ! ~	right to left
2	* / %	left to right
3	+ -	left to right
4	« »	left to right
5	> >= < <= == !=	left to right
6	&	
7	^	
8	|	
9	&&	
10	||	
11	=	

9. Appendix E - The BackTALK Compiler

BackTALK is a compiled script language, which means that scripts must be processed by the BackTALK compiler before they can be used by STalker. Compiling reduces the readable text commands to compressed codes that STalker can interpret. This allows STalker to be much smaller, which is important when it is loaded as a desk accessory. It also allows the compiled scripts to run much faster.

For the technically minded, BackTALK creates true object code. The compiler is single-pass and generates the object code as it parses the source file. The simulated "machine" in STalker that executes the scripts recognizes approximately 50 instructions. Because of this architecture, a completely different language could be built on top of the BackTALK virtual machine. A BASIC-like interpreter could conceivably be written, as could a better version of BackTALK with libraries, or even something more like C or Pascal. Even more likely would be an assembler that allows programs to be written directly for the virtual machine. There are no current plans for such beasts, but they could be done by any programmer possessing the right skills and knowledge of the virtual machine. The compiled scripts are encrypted so that literal strings and constants (e.g. passwords) can not be viewed simply by reading the file.

The operation of the BackTALK compiler, BACKTALK.PRG, is straightforward. When run, the compiler displays a dialog box with two selector buttons, Compile and Quit. The Compile button calls the GEM file selector to choose a BackTALK source file to compile. Processing starts immediately and messages are displayed in the center of the dialog box. If the compile was successful, the script file will be stored in the same path as the source file and the compiler will display a success message. Otherwise, a list of errors will be displayed. The error messages and their explanations are given in Appendix F. The items under the File menu duplicate the buttons in the dialog.

The compiler can be run as a desk accessory or as a GEM program simply by renaming the file with .ACC or .PRG extensions respectively. An advantage of running the compiler as a GEM program is that the desk accessory menu will be available, allowing use of STeno to edit source files and STalker to test compiled scripts. The compiler can also be installed as a *NeoDesk Accessory* (see the section on *Using STalker with NeoDesk* in the *STalker 3* manual). This lets you place the BACKTALK.ACC icon directly on the *NeoDesk* desktop or easy access. Once installed, you can also compile by simply dropping it on the BackTALK accessory icon.

Pressing [Control][Tab] will force the BackTALK Compiler to invoke STalker or STeno, which ever one was invoked it last.

10. Appendix F - BackTALK Compiler Error Messages

- BackTALK file written successfully!
  No errors encountered.
- Identifier previously declared
  A variable or function name was declared or defined twice.
- Invalid array dimension
  A string was declared with a size of something other than a positive integer.
- Invalid character constant
  The closing single quote was omitted from a character constant.
- Numeric constant out of range
  Constants are limited to 32-bit integers. This error also occurs when an invalid number (8 or 9) is used in an octal constant.
- Bad parameter/syntax in function call
  A parameter in a function call is the wrong type or the function call is badly formatted. Insure parentheses surround the parameter list and that commas separate the parameters.
- Bad statement syntax
  Many possible causes. Remember to check that the statement above was ended with a semicolon.
- String constant too long
  String constants may only be up to 512 bytes long.
- Can't open the specified file
  The source file or a header file called from the source file can not be opened.
- Invalid identifier in CONSTANT stmt
  All names must be less than 32 characters, contain digits, alphabetic characters, or '_', and start with an alphabetic character.
- Declaration syntax error
  A variable or function declaration is incorrect.
- '=' expected in constant declaration
  The '=' character was not where it was expected to be in a `CONSTANT INT` declaration.

- Expression expected
  The statement requires an expression that evaluates to an integer value.

- Function type doesn't match declaration
  A function is being defined that was previously declared, and the RETURN data types for the two statements do not match.

- Identifier name too long
  A variable or function name exceeded the 32 character limit.

- INCLUDE statements too deeply nested
  A header file called with an INCLUDE statement may itself have an INCLUDE statement, but calls may only be nested 3 deep. Delete the INCLUDE statements from the last file and place them in the original source file.

- Constant expression required
  A constant expression may only contain numbers, previously defined constants, the `sizeof()` pseudo-function, and operators.

- 'INT' expected in constant declaration
  Constants must be declared with a 'CONSTANT INT' statement.

- Disk error while reading the input file
  The BackTALK source file was not successfully read from the disk.

- Disk error while writing the output file
  The BackTALK object file was not successfully written to the disk.

- '(' expected
  A function call is missing the opening brace for the parameter list.

- Functions were declared but not defined
  Names were declared and space reserved for functions, but the function definitions were not found.

- String constant missing closing "
  Strings must be enclosed with double quotes on both ends.

- Too many functions, out of memory
  The compiler did not have enough memory to hold the symbol table for all the functions declared. The current limit is 128.

- No 'main' function
  Each script must have at least one function defined, and it must be named '`main()`'. Execution of the script will start with '`main()`'.

- Not enough memory available
  The compiler could not allocate the buffer needed to perform the compilation.

- Out of compiler memory
  The total compiled code was larger than the compiler could handle. The current limit is 64k.

- Too many variables, out of memory
  The compiler did not have enough memory to hold the symbol table for all the variables declared. The current limit is 512. Reduce the number of unneeded global variables and constants to make more room.

- ']' expected
  String declarations must have their sizes enclosed in brackets.

- Return value doesn't match function type
  The value of the RETURN statement within a function definition does not match the data type specified by the RETURNS clause at the top of the function definition. The function definition and/or declaration must specify the data type of the value returned by the function. The value actually returned by the 'RETURN' statement must match this data type.

- Variable name required in sizeof()
  The `sizeof()` function is an instruction to the compiler to insert the declared size of the specified variable and requires a name.

- A value can't be assigned to an array
  The script attempted to assign a value to a string.

- Undeclared variable on left side of '='
  Variables must be declared before a value can be assigned to them.

- Undeclared function
  A call was made to a function that was not previously declared or defined.

- Syntax error
  A 'catch-all' error for any case not covered by the messages above.

11. Appendix G - File Operation Error Codes

11.1. BIOS Errors

Code	Explanation
-1	Error
-2	Drive Not Ready
-3	Unknown Command
-4	CRC Error
-5	Bad Request
-6	Seek Error
-7	Unknown Media
-8	Sector Not Found
-9	Out of Paper
-10	Write Fault
-11	Read Fault
-13	Write on Write-Protected Media
-14	Media Change Detected
-15	Unknown Device
-16	Bad Sectors on Format
-17	Insert Other Disk (request)

11.2. GEMDOS Errors

Code	Explanation
-32	Invalid Function Number
-33	File Not Found
-34	Path Not Found
-35	Handle Pool Exhausted
-36	Access Denied
-37	Invalid Handle
-39	Insufficient Memory
-40	Invalid Memory Block Address
-46	Invalid Drive Specification
-48	Not The Same Drive
-49	No More Files
-65	GEMDOS Internal Error
-66	Invalid Executable File Format
-67	Memory Block Growth Failure

11.3. BackTALK Errors

Code	Explanation
-31000	Bad Descriptor (a bad file handle was used)
-31001	Bad Open Mode
-31002	No More Descriptors (no more file handles available)
-31003	End of File
-31004	Read Only (write operation attempted on a read-only file)
-31005	Bad Seek Value

12. Appendix H - Emulations

This section contains a complete list of the control sequences sent and understood by STalker. PC-ANSI emulation mode often abbreviated as "PC"

VT-52 mode controls are generally two character sequences consisting of an ESC character (decimal 27) followed by another character.

VT-100 and PC-ANSI mode controls have 2 main formats.

1. The first form consists of a single 8 bit character, in the range Hex 80 to Hex 9F. These sequences can also be transmitted with 7 bit characteds using the standard code extension technique, which consists of sending an ESC character followed by the command character minus Hex 40. For example, the index command (IND) can be transmitted either as a single character (Hex 84), or as the sequence "ESC D" (ESC Hex 44).

   PC-ANSI mode has printable characters in the range Hex 80 to Hex 9F, so all control sequences must be sent using the 7 bit code extension technique.

2. The second form always begins with the "command sequence introducer" character (CSI). Since this is an 8 bit character (Hex 98), the code extension technique described in the preceding paragraph can also be used, resulting in "ESC [".

   Following the CSI are 0 or more parameters separated by semicolons and terminated by a final character. Each parameter consists of decimal numbers formed by strings of zero or more ASCII digits. The final character is always alphabetic.

   Parameters may be "selective" or "numeric". Selective parameters are used to select one of a set of sub-functions. Numeric parameters explicitly represent a number.

   For example, the sequence "CSI 1 K" will clear the screen from the beginning of the current line up to and including the cursor position. The sequence "CSI 2 K" will clear the complete line. This sequence has a selective parameter that determines which of several functions is invoked.

   The sequence "CSI 10 C" will move the cursor 10 positions to the right. In this case, the "10" is a numeric parameter, since it indicates how many positions the cursor will move.

12.1. Keyboard Generated Sequences

The keyboard produces standard ASCII codes, including all of the control codes fom 0-31. Control codes are generated simply by holding down the [Control] key while striking another key.

12.1.1. BREAK

A BREAK signal can be generated using the [Alternate][Backspace] or [Control][Undo] keystrokes. This will cause a 250 millisecond space condition on the RS232 port.

12.1.2. Carriage Return

The [Return] key will generate either a single carriage return character (decimal 13), or a carriage return and linefeed (13, 10) depending on the state of the new line flag. The new line fag is controlled via the "CSI 20" sequence (see the *VT-100/PC-ANSI Mode Setting Sequence* section).

12.1.3. Cursor keys

The four arrow keys will generate a standard cursor key sequence, depending on the current emulation and cursor key mode (normal or "application").

Key	VT-52	VT-100/PC Normal	VT-100/PC Application
Up	ESC A	CSI A (ESC [A)	SS3 A (ESC O A)
Down	ESC B	CSI B	SS3 B
Right	ESC C	CSI C	SS3 C
Left	ESC D	CSI D	SS3 D

12.1.4. Keypad

The Atari numeric keypad is used by STalker in much the same fashion that a VT-100 terminal uses its auxiliary keypad. Codes sent by the keypad keys depend upon the terminal emulation and the keypad mode (numeric or "application").

Key	VT52 Num	VT52 App.	VT100/PC Num.	VT100/PC Appl.
0	0	ESC ? p	0	SS3 p
1	1	ESC ? q	1	SS3 q
2	2	ESC ? r	2	SS3 r
3	3	ESC ? s	3	SS3 s
4	4	ESC ? t	4	SS3 t
5	5	ESC ? u	5	SS3 u
6	6	ESC ? v	6	SS3 v
7	7	ESC ? w	7	SS3 w
8	8	ESC ? x	8	SS3 x
9	9	ESC ? y	9	SS3 y

-	-	ESC ? m	-	SS3 m
+	+	ESC ? l	+	SS3 l
.	.	ESC ? n	.	SS3 n
Enter	Return	ESC ? M	Return	SS3 M
([F1]	ESC P	ESC P	SS3 P	SS3 P
) [F2]	ESC Q	ESC Q	SS3 Q	SS3 Q
/ [F3]	ESC R	ESC R	SS3 R	SS3 R
* [F4]	ESC S	ESC S	SS3 S	SS3 S

12.2. Received Sequences

The following control sequences are common to all emulation modes:

Mnemonic	Name	Hex	Action
NUL	Null	00	Ignored when received.
BEL	Bell	07	Generates the bell tone.
BS	Backspace	08	Moves the cursor to the left one position. If the cursor is at the left margin, no action is taken.
HT	Horiz. Tab	09	Moves the cursor to the next tab stop. Tabs are located 8 characters apart.
LF	Linefeed	0A	Causes a linefeed or new line operation, depending on the state of the new line flag.
VT	Vert. Tab	0B	Interpreted as a LF.
FF	Form Feed	0C	Interpreted as a LF.
CR	Return	0D	Moves the cursor to the left column on the current line.
SO	Shift-Out	0E	Selects the alternate character set "G1" for all printable character received. This enables accented and other special characters to be displayed over 7 bit communciation lines. The section VT100/PC-ANSI Character Set Selection contains more information.
SI	Shift-In	0F	Selects the normal character set "G0" for all printable characters received. See the description of "Shift-Out" for more information.
CAN	Cancel	18	Aborts processing of an escape or control sequence, otherwise it's ignored.
SUB	Substitute	1A	If received during an escape or control sequence, terminate and cancels

the sequence and causes the sequence and causes a reverse question.

12.3. VT100 Control Codes

Sequence	Name	Hex	Action
IND	Index	84	Moves the cursor down one line in the same column. If the cursor is at the bottom margin, the screen will scroll.
NEL	Next line	85	Moves the cursor to the first position on the next line. If the cursor is at the bottom margin, the screen will scroll.
RI	Rev index	8D	Moves the cursor up one line in the same column. If the cursor is at the top margin, the screen will scroll.
CSI	Cntrl Seq Intro	98	Begins a control sequence.

12.4. VT100 Mode Setting Sequences

Name	Set Sequence	Reset Sequence
Insert/Replace	CSI 2 h (insert)	CSI 2 l (replace)
Linefeed/newline	CSI 20 h (NL)	CSI 20 l (LF)
Cursor key	CSI ? 1 h (app)	CSI ? 1 l (cursor)
VT100/VT52	CSI ? 2 l	See VT52 controls
Auto line wrap	CSI ? 7 h (on)	CSI ? 7 l (off)
Cursor enable	CSI ? 25 h (on)	CSI ? 25 l (off)
Keypad mode	ESC = (app)	ESC > (numeric)

12.5. VT100 Control Sequences

Name	Sequences	Action
Cursor Up	CSI Pn A	Moves the cursor Pn lines up in the same column.
Cursor Down	CSI Pn B	Moves the cursor Pn lines down in the same column.
Cursor Forward	CSI Pn C	Moves the cursor Pn columns right.
Cursor Back	CSI Pn D	Moves the cursor Pn columns left.
Cursor Position	CSI PI ; Pc H	Moves the cursor to line PI, column Pc.

Horiz/Vert Position	CSI PI ; Pc f	Equivalent to "Cursor Position".
Save Cursor	ESC 7	Saves in memory the following information: cursor position, character set shift flag, line wrap flag. VT100 mode only.
Restore Cursor	ESC 8	Restore the information saved with "Save Cursor". VT100 mode only.
Save Cursor	CSI s	Saves the cursor position saved with the last "CSI s" sequence. PC-ANSI
Restore Cursor	CSI u	Restores the cursor position saved with the last "CSI s" sequence. PC-ANSI mode only.
Select Rendition	CSI Ps ; ...	Ps m Selects the video attributes for subsequent characters. When multiple parameters are used, they are processed in sequence and the effect is cumulative. Values for Ps are listed in the "VT100/PC-ANSI Video Attributes" table.
Insert Line	CSI Pn L	Inserts Pn lines at the cursor.
Delete Line	CSI Pn M	Deletes Pn lines starting with the cursor line.
Insert Character	CSI Pn @	Inserts Pn black characters at the cursor position.
Delete Characters	CSI Pn P	Deletes Pn characters starting with the character at the cursor position.
Erase Characters	CSI Pn X	Erases (by changing to black) Pn characters starting with the cursor position.
Erase In Line	CSI K	Erases characters from the cursor to the end of the line.
	CSI 0 K	Same as above.
	CSI 1 K	Erases from the beginning of the line up to and including the cursor.
	CSI 2 K	Erases the complete line.
Erase in Display	CSI J	Erases from the cursor line to the end of the screen.
	CSI 0 J	Same as above.

	CSI 1 J	Erases from the beginning of the screen to the cursor line.
	CSI 2 J	Erases the complete display. In PC-ANSI emulation mode, this sequence also moves the cursor to row 1, column 1 (home).
Set Margins	CSI Pt ; Pb r	Sets the top and bottom "margins" of the scrolling region. Pt is the line number of the first line in the region, and Pb is the is the last line in the region. If either of Pt or Pb is omitted, they default to the top and bottom line of the dispay respectively.
Device Attributes	CSI c	STalker responds to the host query with the sequence "CSI ? 1;2 c", the same sequence returned by a VT100.
	CSI 0 c	Same as above.
Dev. Status Report	CSI 5 n	Operating status query from host. STalker responds to the host query with the sequence "CSI 0 n", indicating that it is operating correctly.
	CSI 6 n	Cursor position query from host. STalker responds with its cursor position using the sequence "CSI Pv ; Ph R" where Pv is the vertical cursor position (row) and Ph is the horizontal cursor position (column).
	CSI ? 15 n	Printer status query from host. STalker responds with the sequence "CSI ? 13 n" to indicate that the printer is not attached.
	CSI ? 25 n	User defined keys query from host. STalker responds with "CSI ? 21 n" to indicate that the keys are locked and may not be changed by the host.

Terminal Reset	CSI ! p	Soft terminal reset - sets emulation values to default states.
	ESC c	Hard reset - has same effect as soft reset.
Tab Stop	ESC H	Request from host to set a tab stop at the current column. Ignored by STalker.
	CSI g	Clear tab stops. Ignored by STalker.
Line Attributes	ESC # 3	Ignored by STalker.
	ESC # 4	
	ESC # 5	
	ESC # 6	
Ctrl Transmission	ESC sp F	Causes all C1 (Hex 80 to 9F) codes to be converted to their equivalent 7 bit extensions. STalker always uses this mode.

12.6. VT100 Video Attributes

Ps	Action
0	All attributes off. Color is not affected.
1	Bold
4	Underlined
5	Blinking (display as 'light' text)
7	Inverse (reverse image)
30	Black characters
31	Red characters
32	Green characters
33	Yellow characters
34	Blue characters
35	Magenta characters
36	Cyan characters
37	White characters

STalker assumes that a standard GEM palette is in effect and will attempt to match the requested color as closely as possible to an available color. When the requested color is not available, an existing color is used and may be combined with the "bold" video attribute. For example, on a 4 color system, MAGENTA will be mapped to BOLD RED.

12.7. VT100 Character Set Selection

Character Set	Sequence	Designate As
ASCII	ESC (B G0 (default)	ESC) B G1
UK National	ESC (A G0	ESC) B G1
DEC Special Graphics	ESC (0 G0	ESC) 0 G1

12.8. VT52 Escape Sequences

The VT52 mode of STalker implements the extended VT52 mode of the Atari ST.

Escape	Function	Comments
ESC A	Cursor UP	
ESC B	Cursor DOWN	
ESC C	Cursor RIGHT	
ESC D	Cursor LEFT	
ESC E	Clear home	
ESC H	Cursor to home	
ESC I	Reverse linefeed	
ESC J	Erase to end of screen	
ESC K	Erase to end of line	
ESC L	Insert line	
ESC M	Delete line	
ESC Y	Line Column	Direct cursor address. The "Line" and "Column" are sent as ASCII codes whose values are the number plus Hex 1F. For example, Hex 20 refers to the first line.
ESC b	Color Set	The character color. The 4 least significant bits of the character following the 'b' are used as the color index into the GEM palette. Colors are re-mapped using the method described in the *VT100 Video Attributes* section. If the color number is between 8 and 15 and the display has less than 16 colors available, the color is mapped to the requested color minus 8. If the new color is available, the 'light' attribute is added, otherwise the 'underlined' attribute is added. For example, on a 4 color system, a request for color number 15 (LIGHT MAGENTA) will be mapped to UNDERLINED BOLD RED.
ESC c	Color Set	The background color - processed but does not affect the display.

```
ESC d   Erase from home to cursor.
ESC e   Turn the cursor on.
ESC f   Turn the cursor off.
ESC j   Save the current cursor position.
ESC k   Restore the saved cursor position.
ESC l   Erase the current line.
ESC o   Erase from the start of the line to the cursor.
ESC p   Reverse on.
ESC q   Reverse off.
ESC v   Enable line wrap.
ESC w   Disable line wrap.
ESC =   Enter alternate keypad mode.
ESC >   Exit alternate keypad mode.
ESC <   Enter VT100 mode.
```

13. Index

A

alert(), 51
appl_find(), 52
appl_id(), 51
appl_write(), 52
ascii_receive(), 53
ascii_send(), 53
ASCII Table, 97
Assignments, 45
atoi(), 54

B

BACKTALK.ACC, 1
BACKTALK.PRG, 1
BackTALK Errors, 112
BackTALK Operators, 105
Binary Operators, 105
BIOS Errors, 111
BITAND, 44
BITNOT, 44
BITOR, 44
BITXOR, 44
BIX, 18
BIXOFF.BTS, 33
BREAK, 114

C

C, 1
Carriage Return, 114
chat_mode(), 54
Comments, 46
Constants, 40
crc(), 54
crc_update(), 54
crc32(), 54
crc32_update(), 55
CSI, 113
Cursor Keys, 114

D

dial_group(), 55
dial_load(), 55
dial_many(), 55
dial_one(), 20, 56
dial_save(), 56
download(), 56

E

Emulations, 113
Error Codes, 111
Error Messages, 108
EXIT, 46
exit_program(), 57

F

file_attrib(), 57
file_change_dir(), 58
file_close(), 58
file_create(), 58
file_delete(), 58
file_dir_first(), 59
file_dir_next(), 60
file_exists(), 61
file_get_dir(), 61
file_get_drive(), 61
file_getchar(), 61
file_gets(), 61
file_instat(), 62
file_open(), 62
file_outstat(, 63
file_printf(), 63
file_putchar(), 63
file_puts(), 64
file_read(), 64
file_rename(), 64
file_seek(), 65
file_write(), 65
FKEY_Load, 22
fkey_load(), 65
FKEY_Save, 22
fkey_save(), 66
FKEYS.BTS, 34
fsel(), 66
Function calls, 46
Function declarations, 47
Function definitions, 47

Functions, 45

G

GEMDOS Errors, 111
get_baud(), 68
get_capture(), 68
get_duplex(), 68
get_emulation(), 68
get_flow_control(), 68
get_int_param(), 69
get_parity(), 69
get_port(), 69
get_printer(), 70
get_protocol(), 70
get_stop_bits(), 70
get_string_param(), 70
get_time(), 71
get_wait(), 30
get_word_length(), 72
getchar(), 66
getkey(), 67
gets(), 67
global_address(), 72

H

HELLO.BTS, 3
Hello World, 3
HELLO2.BTS, 4

I

IF, 48
If-Then, 10
INCLUDE, 48
instat(), 72
Integers, 39
isalnum(), 72
isalpha(), 72
iscntrl(), 73
isdigit(), 73
isgraph(), 73
islower(), 73
isodigit(), 73
isprint(), 73
ispunct(), 73
isspace(, 74
isupper(), 74
isxdigit(), 74
itoa(), 74

K

Keyboard, 9
Keyboard Scancodes, 102
Keyboard Sequences, 114
Keypad, 114

L

Logical signs, 11

M

main(), 18
Main function, 4
memory_read(, 74
memory_write(), 74

N

NULL, 49

O

Operator Precedence, 106
Operators, 43
outstat(), 75

P

Pascal, 1
PC-ANSI, 113
play_sound(), 75
port_break(, 75
port_carrier(), 75
port_connected(), 20, 75
port_drop_dtr(), 76
port_getchar(), 76
port_gets, 16
port_gets(), 76
port_hang_up(), 77
port_instat(), 77
port_outstat(), 77
port_printf, 16
port_printf(), 77
port_putchar(), 19, 77
port_puts(), 77
port_read_string(), 26

port_send(), 78
port_write(), 78
printer_outstat(), 78
printer_printf(), 78
printer_putchar(), 78
printer_puts(), 79
printf(), 5, 79
prot_dtr(), 76
putchar(), 80
puts(), 80

R

Received Sequences, 115
REPEAT, 49
REPEAT-UNTIL, 12, 13
RETURN, 49
run_program(), 81

S

Script
    Compiler, 2
    Interpreter, 2
    Scripts, 37
set_baud(), 81
set_capture(), 81
set_duplex(), 81
set_emulation(), 82
set_flow_control(), 82
set_int_param(), 82
set_parity(), 85
set_port(), 87
set_printer(), 88
set_protocol(), 88
set_stop_bits(), 88
set_string_param(), 85
set_time(), 89
set_word_length(), 89
show_message(), 89
sizeof(), 89
sprintf(), 89
STALKER.BTH, 48
Statements, 45
STeno, 1
steno_append(), 90
steno_id(), 90
strcat(), 90
strchr(), 90
strcmp(), 91
strcpy(), 91
strftime(), 91
String Pointers, 42
Strings, 7, 10, 41
strlen(), 92
strlwr(), 92
strncat(), 92
strncmp(), 92
strncpy(), 93
strrchr(), 93
strstr(), 93
strtoi(), 93
strupr(), 94

T

timer_check(), 94
timer_elapsed(), 94
timer_set(), 94
tolower(), 95
toupper(), 95
Tymnet, 18

U

Unary Operators, 105
upload(), 95

V

Variables, 5
    Assigning values, 7
    Declaring, 6
    Global, 6
    Local, 6
VT-100, 113
    Control Codes, 116
    Control Sequences, 116
    Video Attributes, 119
VT-52, 113

W

wait(), 95
wait_for(), 18, 19, 95
wait_until(), 96
WHILE, 50
While-Endwhile, 12
write_script(), 96